COGENE: AN AUTOMATED DESIGN FRAMEWORK FOR DOMAIN-SPECIFIC ARCHITECTURES

by

Karthik Ramani

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2012

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Karthik Ramani**

has been approved by the following supervisory committee members:

Alan L. Davis	, Chair	**10/26/2012** Date Approved
Rajeev Balasubramonian	, Member	**10/26/2012** Date Approved
John Regehr	, Member	**10/26/2012** Date Approved
Ganesh Gopalakrishnan	, Member	**10/26/2012** Date Approved
Paolo Faraboschi	, Member	**11/05/2012** Date Approved

and by **Alan L. Davis**, Chair of the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

The embedded system space is characterized by a rapid evolution in the complexity and functionality of applications. In addition, the short time-to-market nature of the business motivates the use of programmable devices capable of meeting the conflicting constraints of low-energy, high-performance, and short design times. The keys to achieving these conflicting constraints are specialization and maximally extracting available application parallelism. General purpose processors are flexible but are either too power hungry or lack the necessary performance. Application-specific integrated circuits (ASICS) efficiently meet the performance and power needs but are inflexible. Programmable domain-specific architectures (DSAs) are an attractive middle ground, but their design requires significant time, resources, and expertise in a variety of specialties, which range from application algorithms to architecture and ultimately, circuit design. This dissertation presents CoGenE, a design framework that automates the design of energy-performance-optimal DSAs for embedded systems. For a given application domain and a user-chosen initial architectural specification, CoGenE consists of a a **Co**mpiler to generate execution binary, a simulator **Gen**erator to collect performance/energy statistics, and an **E**xplorer that modifies the current architecture to improve energy-performance-area characteristics. The above process repeats automatically until the user-specified constraints are achieved. This removes or alleviates the time needed to understand the application, manually design the DSA, and generate object code for the DSA. Thus, CoGenE is a new design methodology that represents a significant improvement in performance, energy dissipation, design time, and resources.

This dissertation employs the face recognition domain to showcase a flexible architectural design methodology that creates "ASIC-like" DSAs. The DSAs are instruction set architecture (ISA)-independent and achieve good energy-performance characteristics by coscheduling the often conflicting constraints of data access, data movement, and computation through a flexible interconnect. This represents a significant increase in programming complexity and code generation time. To address this problem, the CoGenE compiler employs integer linear programming (ILP)-based 'interconnect-aware' scheduling techniques

for automatic code generation. The CoGenE explorer employs an iterative technique to search the complete design space and select a set of energy-performance-optimal candidates. When compared to manual designs, results demonstrate that CoGenE produces superior designs for three application domains: face recognition, speech recognition and wireless telephony.

While CoGenE is well suited to applications that exhibit a streaming behavior, multithreaded applications like ray tracing present a different but important challenge. To demonstrate its generality, CoGenE is evaluated in designing a novel multicore N-wide SIMD architecture, known as StreamRay, for the ray tracing domain. CoGenE is used to synthesize the SIMD execution cores, the compiler that generates the application binary, and the interconnection subsystem. Further, separating address and data computations in space reduces data movement and contention for resources, thereby significantly improving performance compared to existing ray tracing approaches.

To my wife, parents, little brother, aunt and uncle, and my grandparents

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Embedded systems have revolutionized the way we interact, perceive, and communicate information. The diverse characteristics of such devices have facilitated their deployment in many areas: inexpensive cellular phones for communication and mobile access to information, reliable pacemakers in the field of medicine, security cameras for surveillance purposes, etc. Recent advances have created a strong market desire for information fusion [76], a broad term that refers to a phenomena in which many different technologies are combined to provide the user with a plethora of usage scenarios. For example, a phone may be used to seamlessly switch between different networks without affecting call continuity and clarity. In the future, a single device will be expected to support many different technologies. While designing such systems presents many new problems to system designers, the following challenges create significant roadblocks.

1.1 Applications

User demand for complex applications and easy-to-use interfaces drives the embedded application space. Providing natural human interfaces requires support for applications like face and speech recognition [55, 28, 47], real-time graphics [18], etc. For communication, a device needs to support a wide variety of cellular standards [42]. The algorithmic complexity of these applications is growing faster than Moore's law [76], but current embedded designs [75] are not flexible enough to adapt to these changes.

Functional fusion, in which one device supports a diverse set of applications, is now a dominant market desire. The iPhone [74] is one example of such a device that provides a few applications, including the touch interface, audio playback, etc. In this case, the functionality for each application is provided by employing a dedicated intellectual property (IP) block on a system-on-chip (SOC) or an application-specific integrated circuit (ASIC). Such IP blocks are specialized circuits that are energy efficient and deliver high performance for one application or a domain of applications with similar computational characteristics.

In the future, devices have to be powerful enough to support an almost ubiquitous set of applications, including video gaming, gesture interfaces, mobile payments, social networking, and traditional desktop applications. Fusing hundreds of ASICs in a single device to support thousands of specialized applications will be practically infeasible due to a variety of reasons, including fabrication costs, yield of the product, etc. However, at any give time, it is likely that the user may only be using a subset of the available applications. This dissertation explores the opportunity to add specialization for an application domain while preserving the flexibility to target a variety of applications within the domain.

The arrival of heterogeneous computing systems like the IBM Cell [38], Intel Larrabee [90], and AMD Fusion [4] has blurred the design requirements in the embedded and desktop computing landscapes. Every new generation of devices is expected to provide an improved level of performance when compared to its predecessor. Power dissipation has also emerged as a first-order design constraint. For mobile devices in particular, energy dissipation should be contained within strict battery life requirements [75]. Unfortunately, battery capacities in mobile devices have been projected to improve at a meager 3-7% [76] every year. Given the exponentially increasing algorithmic complexity, this exacerbates the problem of delivering high performance, low energy, and increased flexibility.

1.2 Nature of Business and Environment

Every new fabrication process requires high initial costs [75, 45] for manufacturing a single chip. Further, millions of chips have to be sold in order to amortize the huge capital investments and to provide sustainable profits. The market need to support new applications every year mandates very short design times for architecting a new chip. In addition, designing an SOC requires large design teams with a variety of expertise, ranging from applications to architecture and circuit design. Thus, the business of embedded devices is governed by extremely short design cycles and economies of large scale [75, 45]. These two constraints are in direct conflict with the amount of time and resources involved in designing and verifying processors in current process technologies. This calls for design methodologies that are scalable and flexible enough to adapt to the volatile markets.

Depending upon the surroundings in which the system is deployed, various constraints have to be met. Data-center computing allows for sophisticated cooling techniques and while power is a concern, performance is given a higher priority. In contrast, small size is important for mobile devices. Easy to use interfaces and ergonomic style are necessary for cellular phones. The deployment environment thus introduces constraints that further

complicate the design of computing systems.

1.3 Traditional Approaches and Drawbacks

Over the last two decades, ASICs were predominantly deployed for embedded computing systems due to their fantastic energy-performance characteristics. This worked well for fixed function devices as ASICs provide a high level of functional specialization while being optimized for area, performance, and power dissipation. Supporting a plethora of complex applications in the future will require lengthy design cycles for each ASIC [45]. In addition, changes in the application will incur expensive redesign costs. Digital signal processors (DSPs) and general purpose processors (GPPs) trade-off energy efficiency to provide flexibility in supporting many applications. They employ a general instruction set (ISA) to support any sequence of operations in a program. The side-effect is that they incur a high control and data access overhead to perform the actual computations. The cost of generality is that they cannot meet the performance and energy requirements for certain applications like face and gesture recognition, cellular standards, and real-time graphics. These applications are characterized by intertwined sequential and parallel code *kernels* phases. While GPPs can deliver good performance for control-intensive sequential code, they incur too much control overhead and power for compute-intensive *kernels* [77]. A good solution is to employ a heterogeneous multiprocessor in which the GPP executes the sequential code and the accelerator executes the various *kernels*. In devices like the iPhone [74], tens of ASICs perform the various *kernel* processing activities, although not all of the applications run at the same time. Hence, the goal of this study is to employ programmable accelerators to replace tens of ASICs.

This dissertation argues that *in a complex design space, automation is the key to satisfying the opposing design themes of high performance, low energy dissipation, flexibility, and short time to market.* Such architectures are referred to as *domain-specific architectures* (DSAs). The DSA [61, 42, 77] is specialized to extract the parallelism within the various kernels of an application domain. For example, the face recognition domain includes all the processes involved in real-time face recognition, including flesh toning, segmentation, face detection, and face identification. A detailed characterization of this domain is performed in Chapter 3. The compute, control, and data access characteristics of all the kernels are analyzed to create a recognition DSA.

The memory system of the DSA consists of hardware support for multiple loop contexts that are common in embedded applications. In addition, the hardware loop unit (HLU)

and address generators provide sophisticated addressing modes which increase IPC since they perform address calculations in parallel with operations performed in the execution units. In combination with multiple SRAM memories, this results in very high memory bandwidth sufficient to feed the execution units. The program is horizontally microcoded and each bit in the program word directly corresponds to a binary value on a physical control wire. This very fine grained VLIW approach was inspired by the RAW project [98]. The side-effect of this microcode approach is ISA-independence and provides the flexibility to mimic the data flow and operations within the program closely while incurring minimal overhead for control flow. Multiple execution units can be chained together to provide "ASIC-like" computation flows due to program controlled data movement through the DSA's resources rather than the usual fetch, decode, and execute microarchitecture. The result is a programmable "ASIC-like" DSA whose energy-delay characteristics approach that of an application specific integrated circuit (ASIC), while retaining most of the flexibility of more traditional programmable processors.

The cost of this microcode approach is increased compiler complexity due to the need to schedule data movement, memory access, register allocation, and execution unit utilization on a cycle by cycle basis. Compile time is also problematic [65, 77], although the compile-rarely nature of these systems mitigates this issue. Another drawback is that it incurs significant time and resources to understand the application domain and design the DSA. For example, the face recognition approach [77] involved man-months of characterization, manual code generation, and architecture design time. Finally, the design of programmable DSAs requires expertise in a variety of specialties, which range from application algorithms to architecture and ultimately, circuit design. To solve these problems, this dissertation presents and explores CoGenE, a single unified framework that automates the design of DSAs for streaming application domains. The goal is to reduce capital costs, time, and resources significantly while meeting the often conflicting system design goals.

1.4 CoGenE: The Grand Goal

CoGenE, which stands for **Co**mpiler-simulator **Gen**erator-design **E**xplorer, is a toolkit intended for use by application domain experts. The automation flow is shown in Figure 1.1. The expert factors the application into sequential code that runs on the general purpose host and *kernel* code that runs on the DSA. In adherence to the stream model employed by the framework, the *kernel* code is modified manually to process data on a per-frame basis and represents streaming code in C. This code is fed as input to CoGenE. The framework

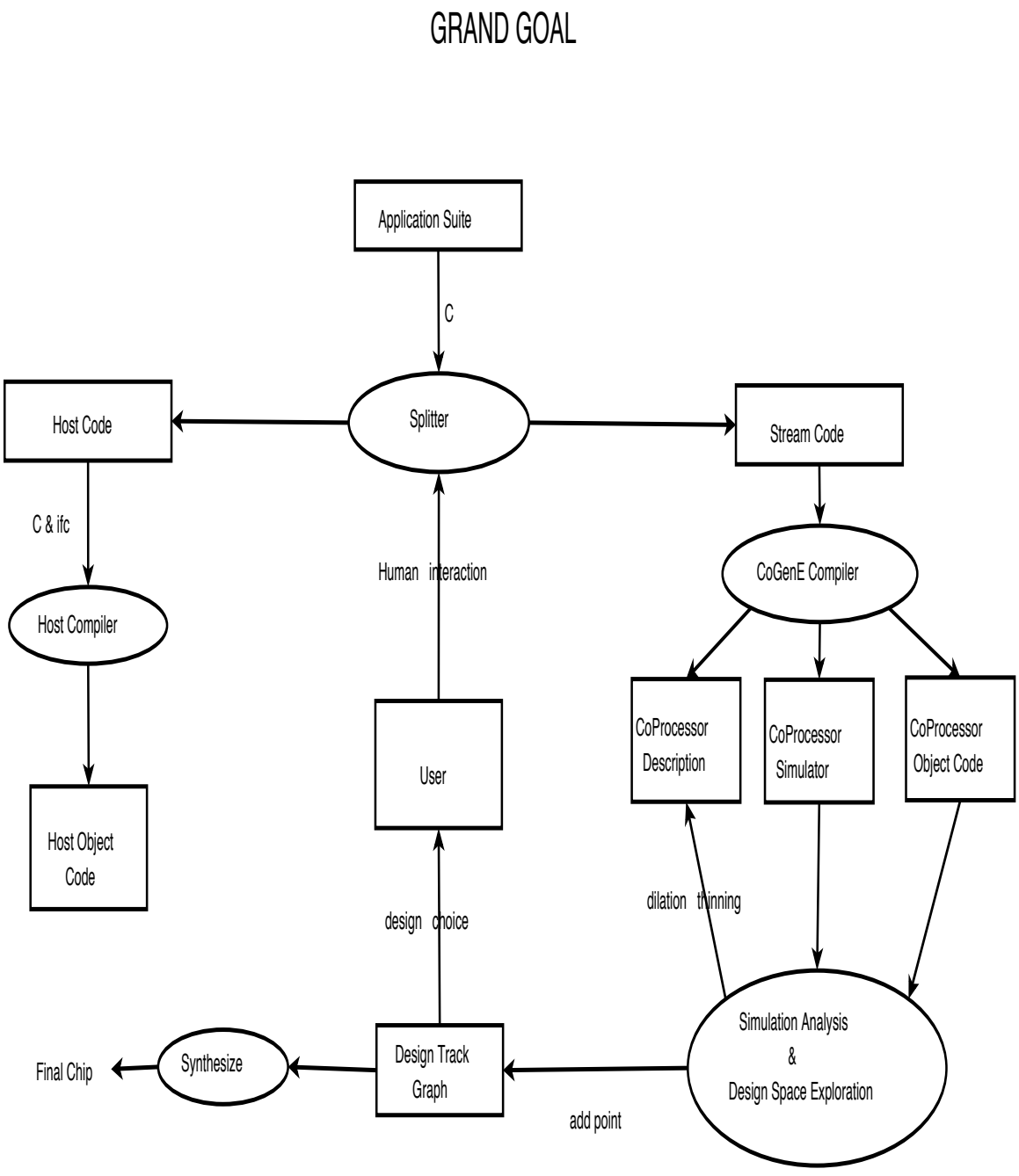

Figure 1.1. Automation from applications to chips

takes the suite that defines the domain, an initial architecture specification of our DSA, and generates a simulator, and an executable binary for that architecture. The architecture is then simulated and both energy, and performance statistics are cataloged. Simulation and compiler data affecting area, energy and performance are then combined. The architecture description is then modified to better satisfy user-specified constraints for any combination of area, power/energy, and performance, at which point the process repeats. Finally, the user is given a set of feasible design points that satisfy his/her requirements. Results (Chapter 8) demonstrate that this process works independent of the choice of the initial starting point and hence, requires little or no architecture expertise from the application expert. The application expert is given a choice to re-factor the code if an adequate solution is not found by CoGenE. In general, CoGenE removes or alleviates the need for compilation, circuit, and architecture expertise, and the error-prone process of designing a specialized accelerator for a given application suite. It also evaluates many more design options than would be possible without similar automation. The result is improved design quality and a significant reduction in design time.

1.4.1 Brief Overview of Framework

CoGenE integrates three distinct activities that all contribute to the design process:

- The compiler that generates execution binary given an architectural specification,
- The simulator generator that creates a cycle-accurate simulator for this template and collects statistics, and
- The design explorer that explores the architectural design space to arrive at energy-performance optimal designs.

The DSA approach employed in CoGenE chains multiple execution units to mimic the data computations within the application with very low overhead. While this delivers high performance, this requires simultaneous scheduling for data motion, function units, and memory accesses in both space and time. To solve this problem, the CoGenE compiler employs integer linear programming (ILP)-based interconnect-aware scheduling techniques to map the kernels to the DSA. The code optimization tactics are based on [77, 78, 36], which have shown that interconnect-aware scheduling improves performance and energy dissipation. After preliminary control and data flow analysis is performed, the innermost loop is identified and memory addressing is set up. After register assignment, ILP-based interconnection scheduling is done followed by postpass scheduling to resolve conflicts.

The resulting object code from the compiler is executed on the simulator. Performance statistics are collected from the resulting compilation and simulation schedules. Energy dissipation is estimated using high-level parameterizable power models. Power models employ analytical models from Wattch [13] for all predictable structures and empirical models similar to [81, 82] for complex structures like the ALU and interconnects. Area estimates are obtained from Synopsys MCL and design compiler. CoGenE's exploration phase then analyzes these statistics to identify potential architectural options for performance or energy improvements.

The simple iterative design space exploration algorithm (Chapter 7) is based on analyzing the source of performance problems that appear during compilation and simulation. Stall causes such as insufficient parallelism, routability problems, etc., all boil down to usage conflicts for various physical resources in the architecture. Adding the appropriate resources (a process called *dilation*) will improve performance but will also increase area and energy consumption. Appropriate removal of lightly or unused resources (*thinning*) may reduce performance but will also reduce energy and area. Improper dilation will increase energy with no performance benefit and improper thinning will significantly reduce performance with little energy benefit. During diagnosis, several options are investigated to remove the bottleneck (this term is used in the context of area and energy as well as the more common performance usage), and each option is assigned a cost. In addition to maximizing performance [32], the notion of cost attempts to optimize energy dissipation and compilation complexity. The least costly alternative is tried first. The process iterates and results in near-optimal designs for user-specified energy-performance constraints.

1.4.2 Evaluation

The effectiveness of CoGenE is evaluated as a case study for three important application domains: face recognition, speech recognition, and wireless telephony. These domains are fundamentally different in their access, control, and computational characteristics [77, 61, 42] and present a diverse embedded workload [55, 28]. The results demonstrate that CoGenE arrives at designs that are competitive with or better than previous best-effort manual designs and significantly better than what can be obtained on more conventional programmable platforms such as the Xscale. The CoGenE compiler generates efficient schedules for a variety of architectures within the DSA framework and performance approaches that of the best manual schedules. The side-effect is that automatic compilation removes the need to invest man-hours into manual code generation. The exploration process is independent of the choice of the initial architectural template and

results show that CoGenE always arrives at optimal energy-performance candidates in a very short time. Overall, this design tool can be employed by application experts to design optimal energy-performance DSAs with little or no expertise in the area of embedded system design.

DSAs designed for embedded applications demonstrate the robust nature of CoGenE for stream-oriented workloads. Workloads that are multithreaded by nature represent a different test to the framework. To evaluate the generality of CoGenE, this dissertation analyzes its capability on ray tracing, a multithreaded graphics application. Ray tracing was chosen due to its many applications in entertainment, science, and industry. In addition, designing an architecture for ray tracing has implications beyond embedded computing.

To fit the CoGenE streaming model, stream filtering is employed. This approach [36] recasts the basic ray tracing algorithm as a series of filter operations that partition an arbitrarily sized group of rays into active and inactive subsets in order to exploit coherence and achieve speedups via SIMD processing. CoGenE is employed to design various constituent parts of StreamRay [80, 79], a novel multicore architecture that efficiently supports ray tracing. The architecture consists of two major subsystems: the ray engine, which performs address computations to form large data streams for SIMD processing, and the wide-SIMD filter engine, which performs the data and filter computations. CoGenE is employed to synthesize the filter engine and the interconnect subsystem. The compiler also generates code for the filter engine. Results demonstrate that StreamRay improves performance significantly and delivers interactive frame rates of 15-32 frames/second (fps) for scenes of high geometric complexity.

1.5 Dissertation Statement

Given the rapidly evolving application space, automation is the key to achieving the opposing design themes of high performance, low energy dissipation, flexibility, and extremely short design time. This dissertation provides the following contributions in achieving these goals:

- **Automation through CoGenE.** A unified design framework that analyzes kernels in an application domain and presents a set of energy-performance optimal DSAs automatically to the application expert who has little knowledge of architecture and circuit design. CoGenE also provides an optimizing compiler that automates code generation. By automating the process of workload characterization, compilation, and architectural design, CoGenE represents a new design methodology that delivers

a significant improvement in system performance, power dissipation, resources, and design time.

- **Workload Studies.** During the initial stage of CoGenE development, the face recognition domain was completely characterized to design a recognition DSA. This is the first study that analyzes the computational requirements of many different face recognition algorithms.

- **CoGenE for Ray Tracing.** This dissertation presents StreamRay, a novel multicore architecture that efficiently supports ray tracing. The CoGenE compiler generates object code for the various ray tracing kernels. The CoGenE explorer was also employed to automatically synthesize the SIMD execution cores and the interconnection subsystem. Given the importance of this emerging application and the new challenges this domain presents to CoGenE, our results demonstrate the robustness of CoGenE in designing DSAs for a variety of compute intensive applications. It also opens a novel area for future work.

1.6 Road-map

A survey of the background work and its limitations is performed in Chapter 2. Chapter 3 showcases our DSA design approach by systematically characterizing and analyzing the face recognition domain. Chapter 4 discusses the various features of our "ASIC-like" DSA methodology followed by the compilation methodology in Chapter 5. Design space exploration is explained in Chapter 7. The evaluation infrastructure and results are discussed in Chapters 6 and 8, respectively. Ray tracing and DSA design is presented in Chapter 9. Conclusions and future work are summarized in Chapter 10.

CHAPTER 2

RELATED WORK

Embedded designs have to achieve the often conflicting goals of high performance, low power, flexibility, and short design time. In recent years, contributions have been made to meet some or all of these goals. The CoGenE design methodology is compared and contrasted against various approaches in the literature to showcase the major differences. It also helps to highlight the novel capabilities provided by CoGenE.

2.1 Applications

2.1.1 Face Recognition

Gottumukkal [35] designed a FPGA-based face recognition (identification) architecture that identifies a person from a database. The difference is that this dissertation performs the study of a complete face recognition system: flesh toning, segmentation, face detection, and face identification. Mathew *et al.* [59] perform a detailed characterization of a feature recognition system based on the Eigenfaces algorithm. In contrast, to our knowledge, this is the first study that compares and contrasts the hardware needs of different recognition algorithms.

2.1.2 Ray Tracing

The use of ray packets to exploit SIMD processing was first introduced by Wald *et al.* [99]. The original implementation targets the x86 SSE extensions, which execute operations using a SIMD width of four, and consequently uses packets of four rays. Later implementations use larger packet sizes of 4×4 rays [7], but these fixed-size packets are neither split nor reordered. Reshetov [84] has shown that even for narrow SIMD units, perfect specular reflection rays undergoing multiple bounces quickly lead to almost completely incoherent ray packets and $\frac{1}{N}$ SIMD efficiency. Thus, worst-case SIMD efficiency is not only a theoretical possibility, but has been demonstrated in current packet-based ray tracing algorithms [11]. Stream filtering in CoGenE maintains high efficiency when processing seemingly incoherent groups of rays,

including secondary rays required for a number of important visual effects. Efficiency is achieved by adding hardware support for filtering divergent rays and by gathering a group of coherent rays for subsequent operation. The evaluation in Chapter 9 demonstrates that it is possible to achieve high SIMD utilization and 50% higher performance while delivering power savings of 12% per SIMD core compared to existing approaches [84].

Several recent research efforts have investigated the problem of coherence in secondary rays. Boulos *et al.* [11] describe packet assembly techniques that achieve CoGenE level performance (in terms of rays/second) for distribution ray tracing as for standard recursive ray tracing. Similarly, Mansson *et al.* [58] describe several coherence metrics for ray reordering to achieve frame rates of 3-5 frames per second (fps) with secondary rays. Instead of tracing rays in a depth-first manner, several works have investigated breadth-first ray traversal. Nakamaru and Ohno [66] describe one such algorithm designed to minimize accesses to scene data and maximize the number of rays processed at a time. Mahovsky and Wyvill [57] have explored breadth-first traversal of bounding volume hierarchies (BVHs) to render complex models with progressively compressed BVHs. This approach, however, uses breadth-first traversal to amortize decompression cost and does not target either interactive performance or SIMD processing. CoGenE builds on these ideas to extract maximum coherence in arbitrarily-sized groups of rays.

2.2 Compilers and Scheduling

Improving performance or power via VLIW techniques is a common theme in modern embedded systems [3], including mapping and instruction scheduling techniques [54, 93]. However, these efforts do not address low-level communication issues. CALiBeR reduces memory pressure in VLIW systems but cannot directly schedule activities to reduce register file communication at the cluster level [2]. Tiwari *et al.* have explored scheduling algorithms for less flexible architectures which split an application between a general purpose processor and an ASIC [95]. Eckstein and Krall focus on minimizing the cost of local variable access to reduce power consumption in DSP processors [29].

Park *et al.* [70] discuss a graph-based software pipelining technique for mapping loops on coarse grain reconfigurable architectures. They have shown performance optimization sacrifices several opportunities for energy reduction. They stress the need for compilation techniques that optimize energy consumption, and employ techniques that significantly reduce energy consumption while minimally degrading performance. High-performance compilation techniques have also been investigated: RAW [53], CGRAs [70], Imagine [85],

and Merrimac [26]. The RAW machine has demonstrated the advantages of low-level scheduling of data movement and processing in function units spread over a two-dimensional space and motivates CoGenE's fine-grained resource control approach. The main difference is that CoGenE's methodology also tries to minimize energy consumption as a first-order design constraint. Mahlke's group has also developed automated techniques for identifying candidate code blocks for coprocessor acceleration and for generating customized instruction set extensions to control those processors [21, 70, 20, 105]. A similar approach by Pozzi also provides graph-based optimizations for micro-architectural constraints such as limited register ports [73]. The main differences between these efforts and CoGenE are the target application space and the approach to co-optimize performance and energy consumption rather than just performance. Results in Chapter 8 show that targeting a DSA for multiple applications [21, 70, 73] consumes significantly higher energy (80%) than targeting a single application domain. Scheduling techniques for power-efficient embedded processors have achieved reasonably low power operation, but they have not achieved the energy-delay efficiency of our architecture [40].

2.3 Embedded Architectures

Recent approaches [17, 65, 70] have proposed the design of programmable processors or coarse-grained reconfigurable arrays for video processing or wireless algorithms for mobile devices. These devices work in various modes to alternatively execute sequential code and the parallel kernels. The problem is that sequential and parallel codes exhibit different kinds of data access characteristics and their execution time varies across different kernels within a domain. For rapidly evolving applications with stringent real time requirements, these devices will be inefficient at extracting different kinds of parallelism and may incur frequent mode changes, thereby degrading application performance.

The MOVE family of architectures explored the concept of transport triggering where computation is done by transferring values to the operand registers of a function unit and starting an operation implicitly via a move targeting a trigger register associated with the function unit [41]. In this dissertation, this concept is used for data transfer between function units.

Application-specific clusters are investigated in [52, 31]. These complementary scheduler approaches minimize inter- rather than intracluster communication and therefore are not able to optimize register utilization as described in this work. In some sense, the fine-grain horizontal microcode approach taken here can be viewed as a fine-grained extension of the

VLIW concept. However, the addition of more sophisticated address generators, multiple address contexts per address generator, the removal of the register file, and the fine-grained steering of data are aspects of this work that are not evident in these other efforts. The energy overhead incurred by the width of the horizontal microcode can be minimized by employing instruction compression or caching techniques [42].

The other parallelism approach that is becoming increasingly popular is short vector or SIMD data parallelism [67, 12]. These techniques have been shown to improve performance by up to an order of magnitude on DSP-style algorithms and even on some small speech processing codes [46]. CoGenE is capable of capitalizing on this form of data parallelism as well. From an energy-delay perspective, however, it was found that SIMD operation [42] does not generally have an advantage. Tensilica's Xtensa system [34], ARM's OptimoDE processor, Bluespec [9], and IBM's Cell processor are all current commercial approaches in the high-performance, energy-efficient embedded systems domain. The main difference is that the user designs a custom VLIW machine by specifying a customized instruction set. In contrast, our ISA-independent approach mimics the data flow within the application closely and significantly reduces the control and access overhead. CoGenE is driven by an application suite and our architecture provides a richer set of options than a traditional more coarse-grained VLIW approach.

2.3.1 Architectural Support for Ray Tracing

Packet-based ray tracing has also been exploited successfully in special-purpose ray tracing hardware projects [89, 103]. We generalize packet-based ray tracing to process arbitrarily sized groups of rays efficiently in wide SIMD environments. While commercial implementations like the G80 [68] and the R770 [4] provide wider-than-four SIMD capability, these machines employ the execution core for address computations and hence, interfere and compete with the actual data computations for resources, thus degrading performance. The Larrabee project [90] employs a many-core task-parallel architecture to support a variety of applications. In contrast, StreamRay extracts performance from ray tracing by efficiently isolating the core tasks of stream generation and stream processing to deliver high performance.

2.3.2 Design Space Exploration

Recent research has investigated exploration techniques [49, 1, 37, 105, 92] to automate the design of application specialized processors or accelerators. Based on the type of architectures explored, these techniques can be classified into three relevant categories. First,

[49, 92] have investigated analytical techniques for the automation of super-scalar processors for SPEC or media application kernels. While [49] is fast, the algorithm was evaluated for one particular program phase, rather than all computational intensive components. [62, 77] have shown that complex multimedia applications like face and speech recognition consists of multiple compute intensive phases. This dissertation employs a multiple context hardware loop unit to efficiently support these phases. While Silvano *et al.* [92] address this issue, their architectural analysis focuses on the memory system and not in great detail on the interconnect and execution units.

The second class of architectures explored for automation is transparent accelerators [105] for embedded systems. This study balances compilation and accelerator constraints and is similar to our approach. While their approach is based on instruction set customization, ours is tailored to extract the data flow patterns within the application. The third and final class of architectures, including our study, fall into the category of long word machines. The PICO design system [1] consists of a VLIW GPP and an optional nonprogrammable accelerator and tries to identify a cost effective combination of the two. Our approach explores the design of a programmable DSA that satisfies the energy-performance-area constraints for the entire application domain.

Grovels *et al.* [24] employed the idea of lost cycles analysis for predicting the source of overheads in a parallel program. They present a tool to analyze performance trade-offs among parallel implementations for a 2D FFT. The CoGenE design employs a similar approach to explore many design points in the architecture space for diverse application domains. Other studies [44] have explored machine learning-based modeling for design spaces and this could potentially replace the simulator employed in our study. In contrast, CoGenE employs static information from the compiler and the integer linear programming scheduler in Chapter 5 to search the design space and arrive at optimal design points for varying constraints.

CHAPTER 3

FROM APPLICATIONS TO ARCHITECTURE

The effectiveness of the framework is evaluated for four different application domains: face recognition, speech recognition, wireless telephony, and ray tracing. The source code for the workloads was obtained from application software research groups in various universities [25, 23, 42, 62]. The applications were manually factored into sequential code and streaming compute intensive *kernels*. Each of the C-based *kernels* were then modified to fit the stream processing model required as input to CoGenE. Mathew *et al.* [62] performed a complete characterization of the speech recognition domain and contributed to the initial architectural methodology. Ibrahim *et al.* [42] characterized the wireless telephony domain. In both cases, manual effort was involved in generating object code for execution on the architecture. This dissertation performs additional detailed characterizations of face recognition and ray tracing domains. In addition, this dissertation presents the design of the optimizing compiler and the explorer that automatically generates the design of the DSA. This chapter begins with a brief overview of speech recognition and wireless telephony. A detailed characterization of the the face recognition domain is then performed to illustrate the salient features of the architectural methodology. The complete process incurred one to two years of design time for one application domain. While time consuming, designs for all three approaches converged to a common architectural approach. This design served as the starting point and led us to explore automation for the process.

3.1 Speech Recognition Overview

The speech recognition application consists of three phases that contribute to 99% of total execution time: preprocessing for feature vector generation, the Hidden Markov language Model (HMM), and the Gaussian (GAU) phase [60]. In preprocessing, sound is represented by Mel-Cepstral vectors [60]. These vectors capture the spectrum of the sound and contain information about the phonemes in sound. In addition, the vectors also

capture the first and second derivatives that contain information about how a phoneme was altered by preceding and succeeding phonemes. The GAU phase, also known as the acoustic model, associates probabilities to the input vectors to map it to a word or series of words in a language. For a given set of input vectors, multiple possible candidates may emerge and they are passed to the final phase. The HMM phase or the language model employs a large table to associate context and meaning to a sequence of words. By interpreting context and meaning, it selects the most probable word sequence.

Preprocessing converts the raw input signal into feature vectors and is dominated by floating point computations. Nevertheless, it only accounts for 1% of the total execution time. GAU and HMM represent Gaussian probability density evaluation and hidden Markov model evaluation, respectively. GAU occupies 57.5% and HMM consumes 41.5% of the execution time of the Sphinx 3.2 speech recognition system. Both Gaussian distributions and hidden Markov models are components of most mature speech recognizers [51, 106]. GAU computes how closely a 10 ms frame of speech matches a known Gaussian probability distribution. One input packet corresponds to evaluating a single acoustic model state over 10 frames of a speech signal. A real-time recognizer needs to process 600,000 invocations of the GAU algorithm every second. The HMM algorithm performs a Viterbi search over a hidden Markov model corresponding to one model state. One input packet to the HMM implementation consists of 32 five-state hidden Markov models. While the GAU algorithm is entirely floating point, the HMM algorithm is dominated by integer compare and select operations. Its average rate of invocation varies significantly with context, but to guarantee real-time performance, it is assumed in this research that all HMM models are evaluated.

3.2 Wireless Telephony Overview

Due to the existence of many different wireless communication protocols [75], the most important *kernels* from signal processing and wireless communication domains are chosen to form a benchmark suite. The matrix multiply, *vec_max*, and the *dotp_sqr kernels* are chosen from the signal processing domain. While *vec_max* selects the maximum amongst a 128 element vector, the dot products $V1.V1$ and $V1.V2$ of two input vectors $V1$ and $V2$ is computed in *dotp_sqr*. The other three applications were selected from the 3G wireless telephony standard [42]. T-FIR is a 16-tap transpose FIR filter. The Rake receiver [42] extracts signals from multipath aliasing effects and the implementation involves four complex correlation fingers. Turbo decoder [42] is a complex encoding application that exhibits superior error correction capabilities. This implementation contains 2 max-log-

MAP modules, an interleaver, and a deinterleaver.

3.3 Face Recognition System Overview

The human face recognition problem is a complex task given the diverse range of facial features and skin tone variations. The importance of face recognition has motivated numerous algorithms [72, 19] and recognition accuracy evaluation efforts [72]. Face recognition can be viewed as two sequential phases: 1) face detection, which analyzes video or camera frames to produce a set of normalized skin-tone patches which likely contain a face, and 2) face identification, which compares the patches to a database of target faces to determine a probable match. Some of the face detection techniques are essentially generalized methods of object detection, and can be adapted to perform other visual feature recognition tasks.

For embedded systems, there is a natural bias towards using cheap, low-resolution cameras. Images may be poorly lit, contain occlusions, and may not contain frontal views. Figure 3.1 shows the major steps involved in face recognition. The input to the system is a stream of 320x200 pixel frames arriving at a rate of 5-10 frames per second. The stream is processed one frame at a time and state is maintained to perform motion tracking. The process is a pipeline of kernels, and the goal is to process them in real time. Flesh toning looks for patches of skin toned pixels. Segmentation looks for a patch that is big enough to contain a face and performs edge smoothing to create a patch. To facilitate processing by the next stage, the patch is converted into a rectangle. Face detection looks for features in the patch which correspond to facial features such as eyes, ears, nose, etc. Eye location pinpoints the probable eye location candidates and normalizes the patch to meet the Face Recognition Technology (FERET) [72] normalization requirements. It also creates a boundary description for the patch. Face recognition then tries to match the probable facial patch to a face in the database. The goal is to minimize the number of false positives and negatives.

The CSU face recognition group has analyzed a variety of face recognition algorithms and has evaluated their accuracy [25, 23]. Two algorithms were chosen due to their superior recognition accuracy and relatively high computational parallelism. The Principal Components Analysis/Linear Discriminant Analysis (PCA+LDA) algorithm recognizes faces by performing holistic image matching while the Elastic Bunch Graph Matching (EBGM) algorithm compares known features (eyes, nose, etc.) of different faces. Because of the fundamental difference in the two algorithms, the execution, data access, and control flow patterns are diverse and together represent a diverse domain. The study in [62] employs

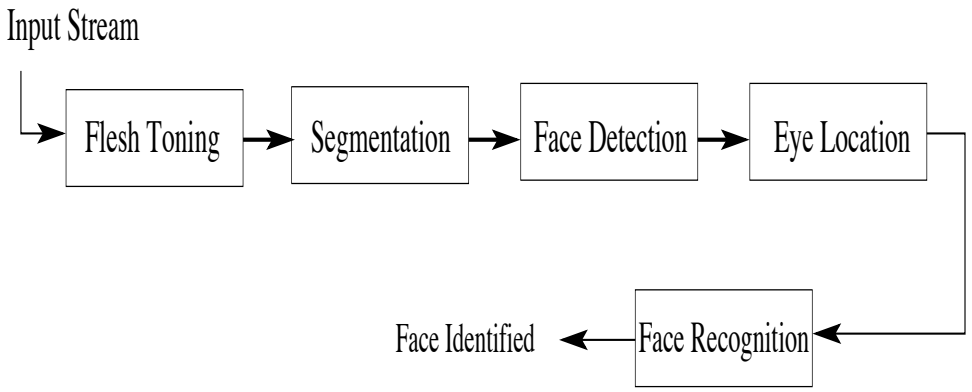

Figure 3.1. Processing Kernels in a Face Recognition System

the PCA technique. A brief description of the different components in a complete face recognition system is followed by a study of the execution profile of the system and its memory requirements. These techniques are also useful in general visual feature and gesture recognition systems.

3.3.1 Preprocessing: Flesh Toning and Segmentation

Skin colors are more tightly clustered in the HSV (Hue, Saturation, Value) or the NCC (Normalized Color Co-ordinated) color space than in the normally employed RGB encoding space. Pixels are thus converted from RGB space to the HSV color space and the NCC space. To improve accuracy, the consensus of two separate flesh toning algorithms based on the NCC and the HSV color spaces are employed respectively [59, 8, 94]. The output of this stage is a bit mask of the image marking where the pixel color is a viable flesh tone.

Image Segmentation is the process of clumping together individual pixels into regions where the face might be found. Because face detection mechanism requires rectangular regions for its operation, two simple mathematical operators are performed: erosion and dilation. An erosion operator examines each pixel and blacks it out unless all its neighbors in a 3x3 pixel map are set [39]. This makes sure that small occlusions are removed from subsequent consideration. Dilation then lights up the pixel if any of its neighbors in a 4x4 window are set.

3.3.2 Viola-Jones Face Detection

The face detector phase is based on the Viola-Jones approach which is similar in purpose to the AdaBoost algorithm [87, 97]. The AdaBoost strategy is to employ a series of increasingly discriminating filters so that weaker/faster filters need to look at larger amounts of data and the stronger/slower filters examine less data. The Viola-Jones takes a

similar approach but rather than cascading filters, their approach is to use multiple parallel weak filters to form a strong filter. Viola-Jones achieves a 15x speedup over the Rowley detector [86]. The Viola-Jones code is proprietary but the algorithm was published and a version of this algorithm was developed at the University of British Columbia (UBC). The AdaBoost algorithm also provides statistical bounds on training and generalization errors. Common operations are sum or difference operations between pixels in adjacent rectangular regions. Face detection involves computing the weighted sum of the chosen rectangles and applying a threshold. A 24x24 detector is swept over every pixel in the image and the image is rescaled. A detection will be reported at several nearby pixel locations at one scale and at corresponding locations in nearby scales. A simple voting mechanism decides the final detection locations. In this approach, a detector with 100 different matching criteria is employed.

3.3.3 Holistic Face Recognition: PCA+LDA Algorithm

Our PCA-based face recognition algorithm is based on [104]. This algorithm was preferred over the Eigenfaces technique [59] due to the increased recognition accuracy in the original FERET study. In the first step, the face images are projected onto a feature space defined by the eigenvectors of a set of faces. The LDA algorithm is then employed to perform image classification. All the training images from the PCA subspace are grouped according to subject identity and basis vectors are computed for each subject. A test image is then projected onto the PCA+LDA subspace and two distance measures are calculated between each pair of images. The distance measures are then used to label the test image for comparison with known persons in the database.

3.3.4 Topology-based Face Recognition: EBGM Algorithm

The EBGM algorithm works on the premise that all human faces have a topological structure and was originally developed by the USC/Bochum group [102]. Faces are represented as graphs, with nodes positioned at facial features such as eyes, nose, etc. and the edges are represented by distance vectors. Distances between the nodes are then used to identify faces. The computational complexity of the algorithm is dependent on the number of feature nodes to be compared. A re-implementation of the EBGM algorithm was provided by the CSU research group [10]. The EBGM advantage is that it performed well in the original FERET studies on facial images that were not frontal views.

The output of eye location is normalized, smoothed, and rescaled in order to increase the efficiency of landmark localization in the face recognition step. The normalized image and

the landmark locations are used to create face graphs for every image in the database. The final step in the algorithm is to produce a distance matrix for the images. Face identification is based on nearest neighbor classification. In the original CSU implementation, real-time performance was not a goal. Hence, the version in this dissertation employs sufficient code motion and reordering to process the image information on a real-time frame-rate basis.

3.4 Workload Characterization

Figures 3.2 and 3.3 show the relative execution profiles for the face recognition system with the PCA/LDA and the EBGM algorithms, respectively. The native profiling results were obtained using SGI SpeedShop on a 666 MHz R14K processor. The face detection kernel accounts for more than 50% and face identification consumes 25% of the total computation cycles. This implies that detection and identification (PCA/LDA and EBGM) are the most time-intensive kernels and are, therefore, the key targets for acceleration.

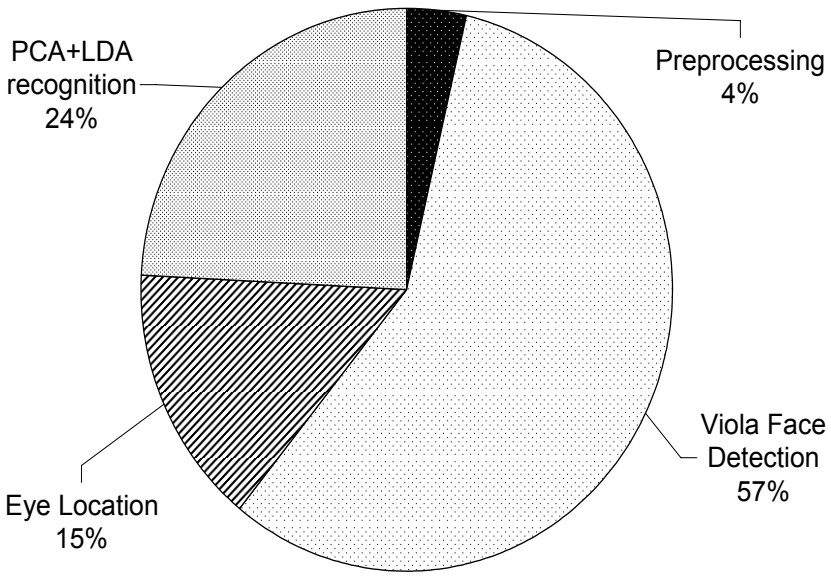

Figure 3.2. Execution profile for PCA/LDA face recognition system

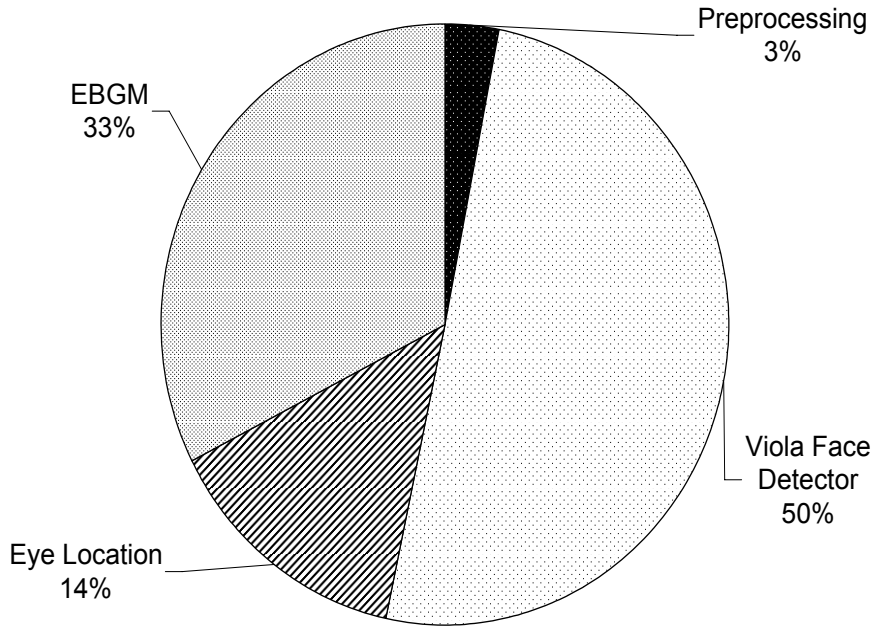

Figure 3.3. Execution profile for EBGM face recognition system

3.4.1 Memory Characteristics

Memory and execution characteristics studies are based on the SimpleScalar [15] simulation framework with architectural parameters chosen to model an out-of-order processor (1.7 GHz) similar to a Alpha 21264. The test configuration is a baseline machine with four integer and four floating point units each in order to provide sufficient execution resources, a 2MB L2 cache, and a 600 MHz DRAM interface. In addition, the size of the caches, the number of integer units, and the number of floating point units are varied for sensitivity analysis.

Figure 3.4 shows the L1 data cache miss rates for four different configurations: i) complete detection pipeline with PCA/LDA identification, ii) complete detection pipeline with EBGM identification, iii) PCA/LDA face recognition without detection, and iv) EBGM recognition without detection. All the configurations achieve 99.4% hit rates in the ICache. We observe good cache locality for all configurations with a small 8KB data cache, which indicates that small self-managed SRAMs are likely to be a good fit for these codes. A 320x200 pixel color image is 188 KB in length while the corresponding gray scale version

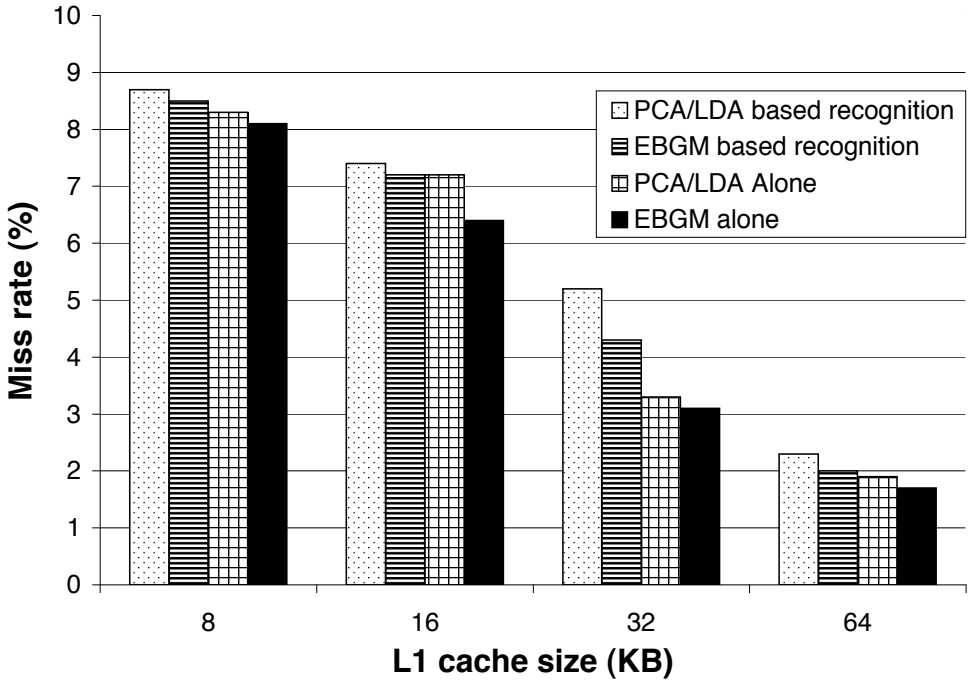

Figure 3.4. L1 cache miss rates

is about 64 KB. While the image will not directly fit in the L1 cache, the flesh toning kernel requires only one pass over every pixel and hence, data can be accessed in a stream based manner. This provides a 64 KB bitmap image that is processed in at most two passes in the segmentation phase. Good cache locality results because the phase accesses at most two rows at a time. Face detection and recognition kernels process even smaller windows (50x50 pixels or 2.5 KB) on this data multiple times and good cache locality is observed for the whole system. Figure 3.5 shows the L2 cache (unified) hit rates for the same configurations. The L2 hit rates are computed as the number of hits in the L2 cache divided by the total number of hits for the application. The very low hit percentages suggest that an L2 cache will be prohibitive in terms of energy and area while providing minimal performance improvements.

3.4.2 IPC Saturation

While the cache behavior of the domain seems to be a good match for embedded processors with limited cache resources, the performance numbers seem to indicate a different view. Table 3.1 shows the instructions committed per cycle (IPC) for four different configurations as the number of integer and floating point function units vary. It can be observed that adding more functional units does not provide a commensurate increase in performance. The configuration with 4 integer and 4 floating point units outperforms the one with 2+2

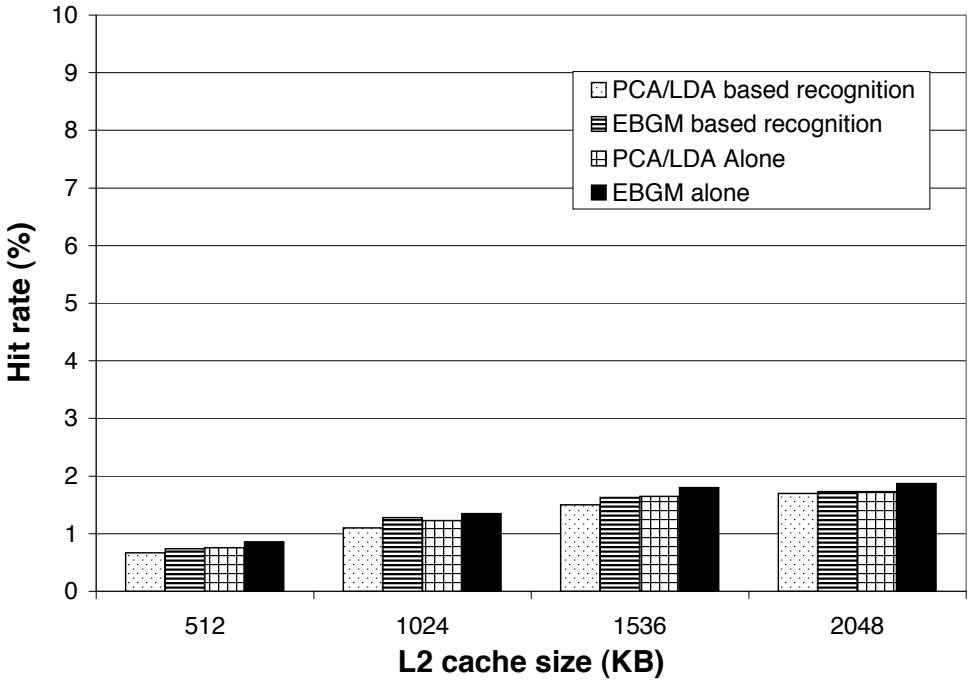

Figure 3.5. L2 cache hit rates

Table 3.1. Instructions per Cycle (IPC) for baseline alpha configuration with varying number of execution units (XUs)

Num. XUs	PCA/LDA complete	EBGM complete	PCA/LDA alone	EBGM alone
1+1	0.651	0.623	0.780	0.757
2+2	0.703	0.683	0.830	0.793
3+3	0.727	0.712	0.897	0.877
4+4	0.729	0.720	0.905	0.890

units by a marginal 5%. In addition, performance saturates beyond 6 units (3+3).

Table 3.2 shows the speedup or slowdown of the four configurations over actual real-time corresponding to 5 frames per second. It can be observed that executing a complete face recognition application is at least 2 times slower than real-time with less than 2+2 functional units. At best, the applications run 1.78 times slower than real-time by adding more resources. Executing the identification algorithms alone can achieve real-time performance with sufficient resources. The performance improvement comes at the cost of a significant increase in power dissipation. The power dissipated by an out-of-order core like the Alpha is likely in tens of watts and this exceeds the power budgets available for embedded systems. This motivates the search for a non-GPP approach to provide real-time face recognition at

Table 3.2. Speedup/slowdown over real-time corresponding to 5 frames per second (real-time is scaled to 1)

Num. XUs	PCA/LDA complete	EBGM complete	PCA/LDA alone	EBGM alone
1+1	2.310	2.560	1.530	1.610
2+2	2.050	2.107	1.378	1.383
3+3	1.800	1.870	1.040	1.160
4+4	1.780	1.784	0.978	1.003

power levels compatible with the embedded space.

There are four reasons for the low performance. They are summarized below:

- The face recognition kernels commonly perform a lot of computations of the form $Z[i] = Z[i-1] + \sum_{j=0}^{m} X[j] * Y[W[j]]$, which contains loop carried dependencies.
- The problem is further exacerbated in multilevel loops where such computations entail complex indirect accesses.
- A large number of loop variable accesses compete with the actual array data accesses, causing port saturation in the data cache. Since the ratio of array variable accesses is high compared to the number of arithmetic operations, contention is a big issue.
- The slow real-time rate indicates that instruction throughput is low. Even when functional units are available, dependencies and memory contention significantly reduce the actual IPC.

3.5 Architectural Implications

Increasing the number of SRAM ports in the system can address the problem of port saturation. Given that an 8KB cache provides good locality in a conventional cache-based system and the L2 miss rate is high, this motivates a choice to use self-managed SRAMs. Three distributed 8KB SRAMs (input,output, and scratch) were employed for the face recognition DSA. The input and output SRAMs can be double-buffered to allow simultaneous communication with the host and the execution cluster. The scratch SRAM is used for holding intermediate data. In addition, each SRAM is dual ported to support the needs of the multiple execution units. The system mimics a distributed 24KB cache with 6-ports but does so more efficiently in terms of area, power, and latency.

3.5.1 DSA Memory Architecture

As with most real-time applications, face recognition loops run for a fixed number of iterations and loop indices are used in data address calculations. The predominant data access pattern consists of 2D array and vector accesses. Extracting parallelism across multilevel nested loops requires complex addressing modes. A hardware loop unit (HLU) is a programmable hardware structure that provides support for multiple simultaneous loop contexts for efficient data access . The loop unit automatically updates the loop nest indices in the proper order and the implementation is similar to [62]. The Viola/Jones detection kernel requires a maximum of three simultaneous loop contexts. Hence, the loop unit supports 3 contexts. Increasing the number of contexts further increases the area, complexity, and power dissipation while providing little performance improvements for the face recognition domain. In addition, the loop unit provides hardware support for modulo scheduling.

The problem of contention between address calculations and actual data computations is only partially solved with distributed memory. The use of programmable Address Generator Units (AGUs) on each SRAM port allows multiple address calculations to be done in parallel with arithmetic operations, which improves IPC. Each AGU effectively services the needs for a particular pipeline. The AGUs use the index values provided by the loop unit to facilitate data delivery to the execution units. Overall, the memory system for the DSA consists of a loop unit, three distributed 8KB SRAMs with two ports each, and associated AGUs.

3.5.2 Execution Back-end: "ASIC-like" Flows

In a traditional super-scalar processor, instructions are fetched, decoded, issued, and retired. Function units receive operands from a register file and return results to the register file. This represents a huge amount of overhead, which then gets amortized over over relatively miniscule amount of computation work in the function unit. The challenge is to amortize the overhead over more work in order to increase performance and reduce power consumption. ASICs are complex computational pipelines which transform input data into results with almost no overhead, but they lack generality and flexibility. Our execution back-end mimics the ASIC approach while preserving programmability. The use of programmable multiplexers allows function units to be linked into 'ASIC-like' pipelines which persist as long as they are needed. The outputs of each MUX stage and each execution unit is registered, which allows value lifetime and value motion to be under program control. This removes the need for a large multiported register file, which saves significant power

with no reduction in performance. Flexibility is preserved by providing the ability to specify interconnect routes via MUX configurations under program control.

The execution resources need to support a large amount of floating point calculations in the face recognition kernels. In addition, integer arithmetic is also required to support address calculations in cases where the AGUs cannot handle these duties autonomously. Our execution units comprise four floating point units and three integer functional units. As will be seen, this provides a good balance between performance and energy consumption.

A SIMD approach also delivers high data parallelism and reduces register file complexity by clustering the register file and thereby reducing port complexity. Our VLIW approach provides high instruction level parallelism by performing memory operations and data computations simultaneously, albeit with a larger control overhead due to the width of the instruction word. Our execution back-end is less dependent on a centralized register file. Moreover, the vast difference in the type of data and address computations performed in a cycle in the face recognition domain makes the SIMD approach less efficient. From performance and energy perspectives, a VLIW approach is more beneficial and is the choice for face recognition.

During the course of characterization of various application domains, a few trends emerged that motivated us to explore automation. First, most of these applications are characterized by streaming data. An input frame is read once, an output frame is written once, and little data are used for preserving state across frame boundaries. This led to the evolution of a unified memory design approach for the three domains. Second, energy efficient execution dictates that the function units chain operations and maximize the compute to access ratio. Automation would save a significant amount of time (6-7 man years for three domains) while allowing us to explore many candidate design choices.

CHAPTER 4

DSA SYSTEM ARCHITECTURE

At the system level, this dissertation employs a heterogeneous multiprocessor and comprises a general purpose processor (GPP) for sequential code and a DSA to accelerate the kernels. The host GPP can be an ARM or x86 CPU or a digital signal processor (DSP) core. The architecture, depicted in Figure 4.1, is an example of a decoupled access-execute architecture [69]. The host GPP handles general control and set-up duties and moves data to and from the DSA via double buffered input and output SRAMs.

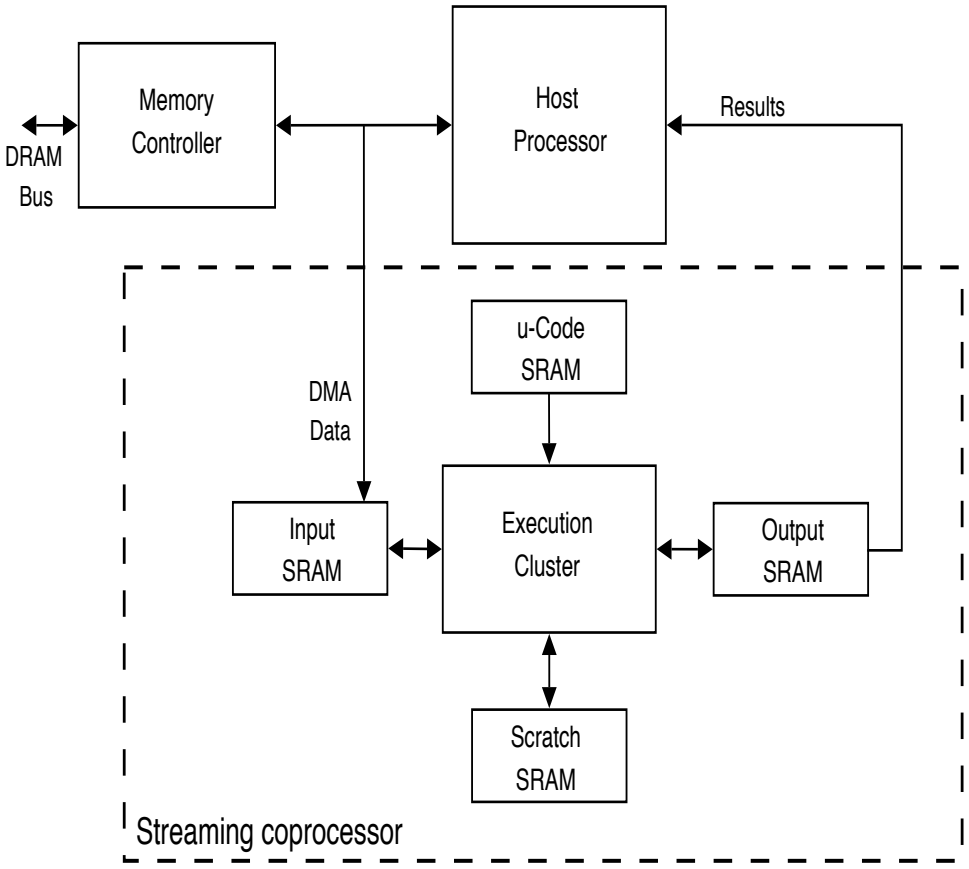

Figure 4.1. Heterogeneous Multiprocessor Organization

The DSA is shown in Figure 4.2. The memory system includes a HLU, SRAMs, and address generator units (AGU). Each HLU context stores the current value of the loop variables in a kernel's loop nest. If multiple kernels will be concurrently active, then multiple contexts are necessary to avoid delays in reloading context data into the HLU. The loop variable values are used by the AGU's for generating addresses to support various addressing modes, including 2D array accesses for row and column walks, strided and strided offset accesses, and complex patterns including A[B[i]] [61].

The use of multiple SRAMs provides higher memory bandwidth. Each SRAM is role-specific in this stream-based DSA strategy, in which applications consume input frames to produce output data and state information for subsequent frame processing. Since the input SRAM is double buffered, the host processor loads the next input frame while the DSA is processing the current frame. The output SRAM is similarly structured so the host processor can remove the previous frame outputs while the DSA is generating the current frame outputs. The scratch SRAM may be dual ported, but in this case, both ports would

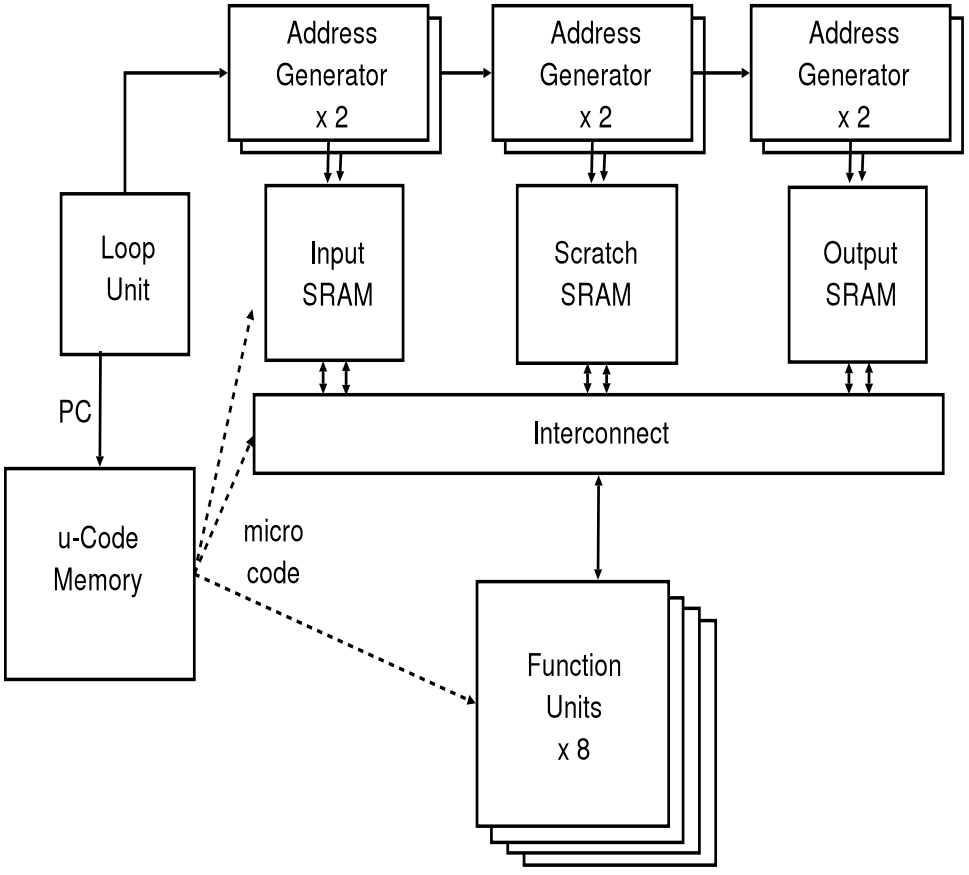

Figure 4.2. Organization of the Recognition DSA

be used by the DSA in order to increase state data bandwidth. The HLU permits modulo scheduling [83] of loops whose loop counts are not known at compile time and this capability reduces compilation complexity.

The horizontal microcode approach allows the multiplexer-based interconnect to be configured under program control (Figure 4.3). This allows function units and their associated pipeline registers to be linked to create pipelines, which persist for as long as they are needed. This persistent pipeline characteristic is similar to the fixed yet inflexible pipelines found in application-specific integrated circuits (ASICs) and is a significant factor in the energy-delay efficiency of the approach. Value lifetime and motion are also under program control.

The compiler generated microcode controls data steering, clock gating (including pipeline registers), function unit utilization, and single-cycle reconfiguration of the address gener-

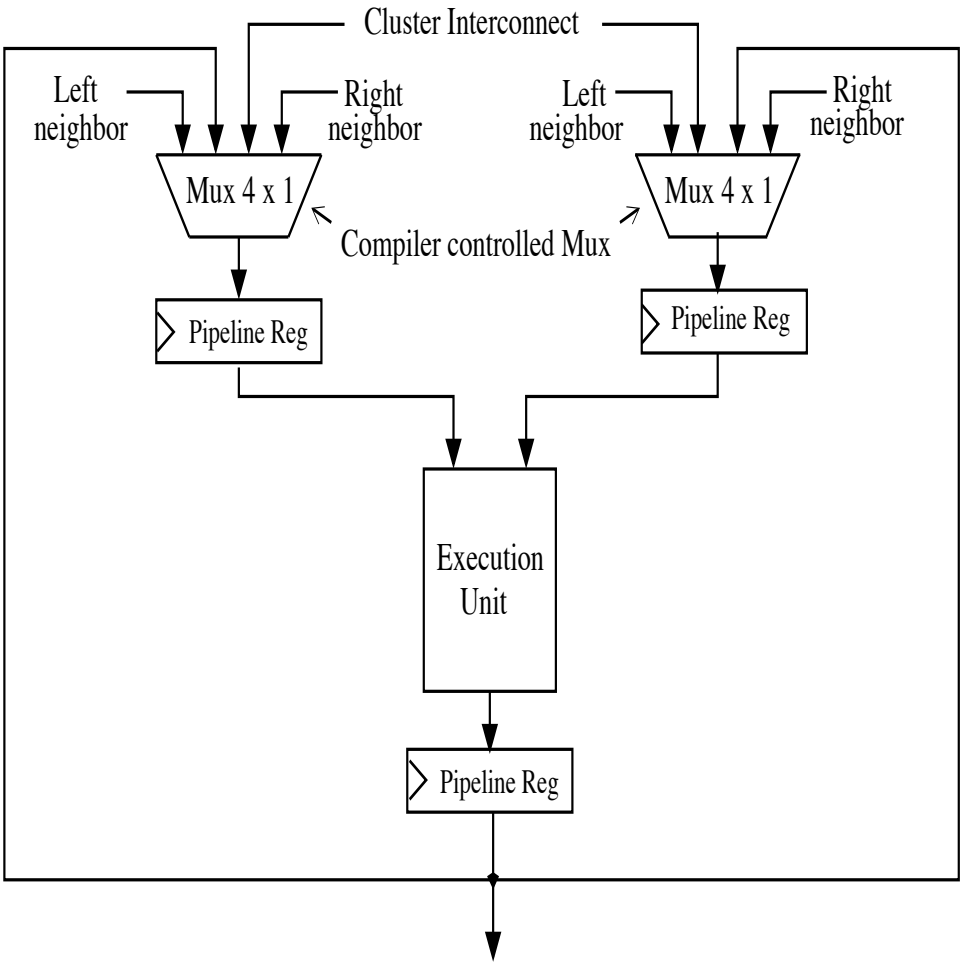

Figure 4.3. Functional Unit Architecture

ators associated with the SRAM ports. A functional unit can either be an integer or floating point execution unit or a register file. As with any highly parallel system, the interconnect subsystem is performance critical. Operating frequency can be increased by reducing individual multiplexer widths and/or adding additional multiplexer levels. The result is improved interconnect throughput at the cost of a slight increase in fall-through delay.

4.1 DSA Evaluation for Face Recognition

Figure 4.4 compares the IPC of the baseline Alpha machine with different DSA configurations:

- DSA with perfect back-end implies no stalls due to communication or execution resources, which shows the performance of the memory system,
- DSA with perfect memory system, which indicates the performance of the interconnect and execution cluster back-end,
- Complete DSA configuration with actual memory and back-end, but with seven functional units and the register file, and
- Complete DSA configuration with eight functional units and no register file.

It can be observed that the DSA configuration with perfect back-end provides as much as a 4.5x IPC improvement for face detection, and around a 10x IPC improvement for face identification (EBGM and PCA/LDA) over a general purpose processor with a traditional cache architecture. This shows that the memory system reduces port contention significantly and efficiently supports indirect addressing schemes.

The configuration with perfect memory evaluates the cluster back-end. When compared to the Alpha processor, this configuration provides a 3x improvement for face detection and 6.7x improvement for face identification. The advantage comes from exploiting persistent pipeline flows where scheduling data for high computation to storage ratio sustains the high memory bandwidth inherent in the system. It also serves to demonstrate the effectiveness of the pipelined registers for storing intermediate values.

The last two configurations in Figure 4.4 show the performance of the complete DSA with the actual memory and actual execution cluster. Here, the performance of the system with and without a register file is done in order to evaluate the effectiveness of the register file. In addition, the register file is replaced by an integer functional unit to evaluate performance

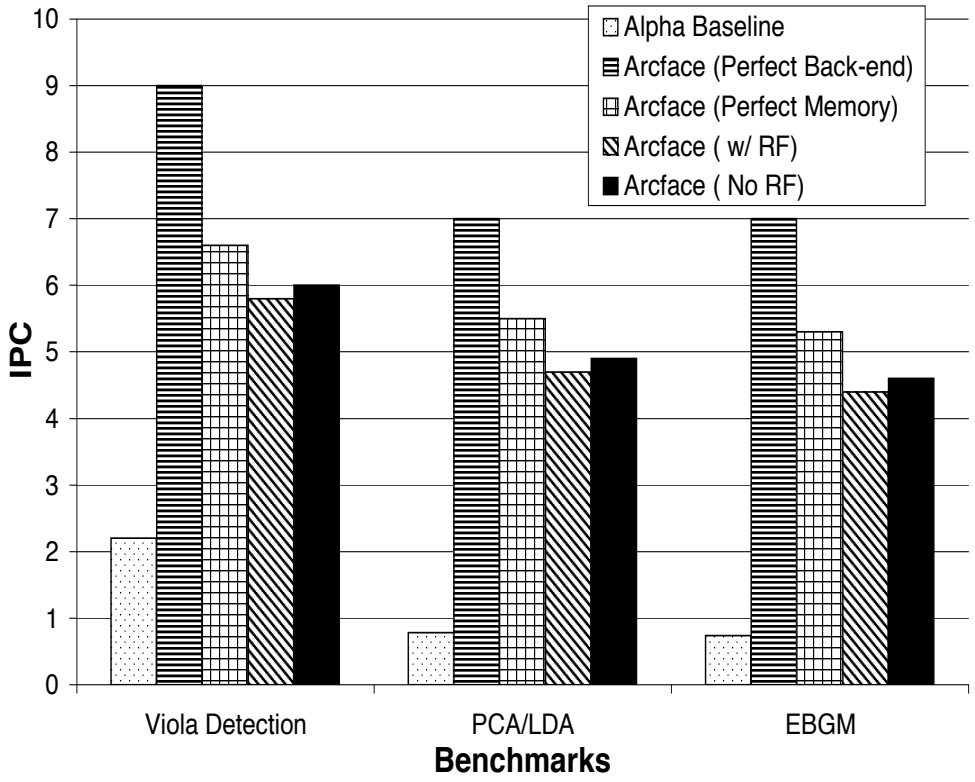

Figure 4.4. Plots showing the potential for memory parallelism and 'ASIC-like' flows

trade-offs. The complete DSA provides as much as a 2.7x performance improvement for face detection and a 5.5x-5.8x improvement for the face identification kernels when compared to the Alpha. The execution cluster and memory system are well matched in terms of throughput. The combination of high memory parallelism and "ASIC-like" flows works well for the face recognition domain. Replacing the register file with an additional integer functional unit provides a marginal 3-4% performance improvement. The register file does ease the difficulty of compiler-based scheduling and is a more generally useful structure than another execution unit if the algorithms change in a substantial fashion.

Comparing the complete model to the model with perfect memory shows a performance degradation of about 13-18%. This is explained by the fact that the baseline system employs a cluster-wide interconnect for communication between the memory and the execution units. Due to contention in the global interconnect for data computation and data access, scheduling delays are introduced, leading to a performance degradation. Employing a hierarchical or separate interconnect will solve the problem, but at increased power costs. Given the performance goal of meeting real time requirements, the power conservative choice is chosen.

The fine-grained horizontal microcoded nature of the DSA implies that the compiler is responsible for managing all of the physical resources at an equally fine-grained level. Managing different function units, multiple memories and their associated AGUs, and scheduling data flows through the interconnect is a complex task. The inherent programming complexity of the architecture makes hand coding a lengthy and error-prone process. Even though the architecture is capable of impressive performance at low power consumption levels, it will be a futile effort unless the scheduling task can be performed automatically by a compiler. The CoGenE compiler that alleviates code generation time is described in Chapter 5. This is followed by a discussion of the CoGenE design space explorer in Chapter 7.

CHAPTER 5

THE COGENE COMPILER

The architectural flexibility of the DSA lends itself to be tailored to satisfy the performance and energy demands of the application as it evolves over many generations. This process of tailoring requires that the application expert communicate the workload requirements to the compiler designer and architect, who then agree on the final design of the chip. In a market with short time-to-market constraints, this process is prohibitive. Ideally, the application expert would like to employ a tool to automatically design the architecture and generate the associated software tools required to run the application on an architecture simulator to get an estimate of performance and energy consumption. This dissertation presents CoGenE, a tool that solves the above problem for the application expert while requiring minimal to no knowledge of the intricacies of compiler, architecture, or circuit design.

"ASIC-like" DSAs deliver high performance due to the ability to coschedule data computations and address computations in space and time on the programmable interconnect on a cycle-by-cycle basis. This improves the computation to access ratio and energy dissipation is reduced as a result of minimized data movement. This dissertation proposes and employs a novel interconnect scheduling phase to produce optimized code for the DSA. The effectiveness of the CoGenE compiler in reducing code generation time while delivering high performance for the recognition domain is also discussed.

5.1 Trimaran to CoGenE

The Trimaran compiler (www.trimaran.org) was the starting point for the CoGenE (Compiler Generator Explorer) compiler development. Trimaran was chosen since it allows new back-end extensions, and because its native machine model is VLIW [88]. Significant modifications were needed to transform Trimaran from a traditional cache-and-register architecture to meet the needs of the DSA's fine-grained cache-less approach.

The result is a compiler that takes streaming code, written in C, and code generation is parameterized by a machine description file which specifies: the number of clusters, the number and type of functional units in each cluster, the number of levels of intercluster and intracluster interconnect, and the individual multiplexer configurations. A new back-end code generator that is capable of generating object code for the coprocessor architecture described by the architecture description file was developed. The code generator includes a modified register allocator that performs allocation for multiple distributed register files rather than for a single register file. Since the compiler controls the programming of the multiplexers and the liveness of the pipeline registers, register allocation is inherently tightly coupled with interconnect scheduling. Hence, a separate interconnect scheduling process is performed after register allocation and the scheduling scheme is based on integer linear programming (ILP) [64] techniques. Before delving into the scheduling details, an overview of ILP-based problem solving is provided.

5.1.1 Integer Linear Programming (ILP)

Computing an optimal solution for an ILP program is NP complete [30]. Researchers at Saarland University have contributed to significant advances in improving the efficiency of ILP techniques by reducing the process of enumeration [30]. Integer Linear Programming is the following optimization problem:

$$min\ z_{IP} = c^T x$$

$$x \in P_F \cap Z^n$$

where

$$P_F = \{x | Ax \geq b, x \in IR^n_+\}, c \in IR^n, b \in IR^m, A \in IR^{mxn}$$

z_{IP} is the objective function that needs to be optimized subject to a set of constraints. The set P_F is called the feasible region and it is integral if it is equal to the convex hull P_I of the integer points ($P_I = conv(\{x | x \in P_F \cap Z^n\})$). In this case, the optimal solution can be calculated in polynomial time, and hence, any formulation of the ILP program should find equality constraints such that P_F is integral.

5.2 CoGenE Compiler Flow

The overall CoGenE flow is illustrated in Figure 5.1. The Trimaran loop detection analysis package is used to identify the loops and calculate the start and end conditions. The standard Trimaran data flow packages are used to annotate the dependence graph with

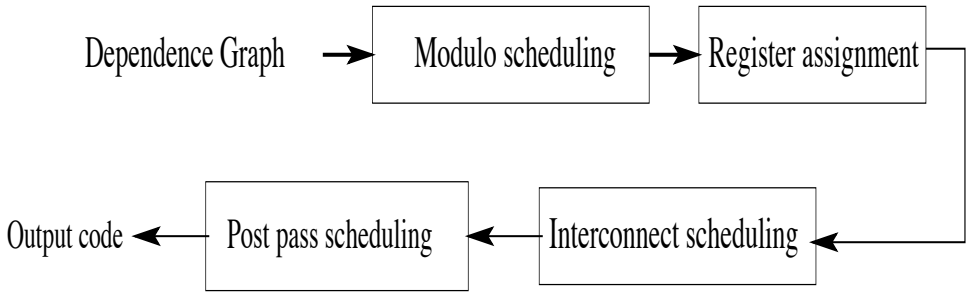

Figure 5.1. Code Generation

variable use and definition locations. Back substitution is then performed to reduce critical path length. After this stage, the number of loops and their characteristics are known.

5.2.1 Modulo Scheduling

With information from the previous step, the innermost loop and the lowest bound on the initiation interval are computed, similar to the modulo scheduling approach [83]. If the bound is high enough to cause degradation, loop unrolling is performed to improve the results of scheduling. This is followed by a simple register assignment scheme where the pipeline registers hold the result.

5.2.2 Interconnection Scheduling

The main decision variables employed are x_{nt}^k where a value of 1 means that instruction n is executed in clock cycle t on execution unit k. The index k of the decision variables is relevant for instructions that can be executed on several different execution units. For all address calculations, the AGUs are paired to a unique execution unit. Let I denote the set of instructions from the input program. The interval N(n) is the earliest control step in which instruction n can be started without violating any data dependencies.

The scheduling polytope is composed of different types of constraints. The assignment constraint ensures that each instruction is executed exactly once by one execution resource. Let R(n) denote the set of execution unit types that the instruction n can be assigned to:

$$\sum_{k \in R(n)} \sum_{t \in N(n)} x_{nt}^k = 1 \quad \forall n \in I$$

The precedence constraint models the data dependencies within the input program. The dependences can be further classified into two categories: weak or antidependences (Write after Read), and strong dependencies (Read after Write). Write after Write dependencies are not an issue in this architecture since write targets do not conflict. Weak dependencies

within a group are allowed. Let w_{mn} represent the minimum number of cycles from start time m to end n during which the dependence is to be respected, then:

$$\sum_{k} \sum_{t_n \leq t} x^k_{nt_n} + \sum_{k} \sum_{t_m \geq t - w_{mn} + 1} x^k_{mt_m} \leq 1$$

The precedence constraints exclude any ordering of instructions where data dependences are violated. Until now, the feasibility function is integral, i.e., the solution can be calculated in polynomial time. Resource constraints are now added to the system. Resource constrained scheduling is NP complete. Let R_k denote the number of execution units of type k available in the processor. The resource constraint prevents more than R_k instructions being assigned in a cycle. It should be noted that resource constraints also implicitly include the constraints on the multiplexer at the output of the execution units. If U is the precalculated upper bound on the number of clock cycles for the input program, then:

$$\sum_{n \in I : k \in R(n)} x^k_{nt} \leq R_k \quad \forall k \wedge 1 \leq t \leq U$$

Now, every integer point saturating the constraints corresponds to a feasible solution of the interconnect scheduling algorithm. The goal is to find a schedule of minimal length L. The value of L is defined by:

$$\sum_{k} \sum_{t \in N(n)} t x^k_{nt} \leq L \quad \forall n \in I$$

The goal is to minimize the objective function L. So far, our objective function does not take into consideration the instructions that take several clock cycles because of interconnect constraints. This could produce instruction slots with no instructions to be scheduled. The objective function minimizes the execution time as a primary constraint. The ILP model in the infrastructure is a solver that employs the simplex method and a solution was efficiently obtained within minutes for most kernels.

5.2.3 Postpass Scheduling

A final pass is done over the code and conflicts in scheduling that can happen due to weak dependencies are distributed to the register file. In addition, those resources that are not used are completely clock gated when their instruction slots are empty. For modulo scheduled loops, a check is made to see if the loop and the address contexts are correctly programmed with the initiation interval.

5.2.4 Efficiency of Interconnect-aware Scheduling

The efficiency of interconnect-aware scheduling is estimated by comparing it against hand-coded schedules. One metric that is useful is utilization rate, a measure of the total fraction of time for which all the seven functional units in the DSA are employed. Table 5.1 shows that we observe around 62-65% utilization rate for the PCA/LDA and the EBGM face identification kernels. Overall, the compiled code achieves an average utilization rate of 60%. When compared to manual scheduling that incurs man-months of optimization for each kernel, the compiler delivers close to 85% of the actual performance within minutes. The 15% disparity is because weak dependencies introduce conflicts in scheduling and this causes delays in the compiled code. Further, data transfers across functional units that are spatially further away from each other incurs longer delays. Addressing these issues will improve the scheduling algorithm; however, our technique still delivers a high utilization rate. The high utilization rates also demonstrate the effectiveness of interconnect-aware scheduling for delivering high instruction throughput. CoGenE incurs tens of seconds of compilation time for all the kernels except for the EBGM kernel in which ILP solving incurs hours to a few days to explore a few feasible schedules from a large scheduling space. When compared to man-months of manual code generation, the CoGenE compiler provides a significant reduction in design time. Further, CoGenE's modularity in compiling to many different architectural templates helps the application expert in exploring a variety of design options.

Table 5.1. Functional unit utilization rate and compilation time for the different face recognition kernels

Benchmarks	Utilization rate (Compilation)	Utilization rate (Manual)	Compilation time (seconds)
Flesh Tone	0.57	0.74	23
Erode	0.575	0.675	37
Dilate	0.570	0.65	40
Viola	0.69	0.75	60
PCA/LDA	0.62	-	49
EBGM	0.65	-	$\geq$1000

CHAPTER 6

THE COGENE SIMULATOR GENERATOR

The CoGenE compiler generates executable binary for various architectural configurations as specified by the architecture template file. Along with the code generator, the framework also generates a cycle-accurate architectural simulator that can be used to collect program statistics. Performance estimation is similar to existing cycle-accurate simulators like Simple-Scalar [16]. Power dissipation is also a first-order design constraint and hence, this dissertation presents an architectural power estimation framework [81] that employs the combination of two different models. We employ analytical models for regular and predictable structures like memory, FIFOs, etc. Power dissipation for complicated structures like execution units, control logic, etc., depend greatly on the implementation and hence, we employ empirical models based on low-level RTL-based power models [81]. Interconnect power dissipation contributes to a significant fraction of total chip power [56] and hence, an area cost is used to build models for interconnects based on the methodology described in [6, 100].

6.1 Simulation: Power and Energy Estimation

Early stage power estimation for CPUs has been a popular research area in the academic community. Wattch [14], a power simulator employs parameterizable analytical models of units like memory structures, clock tree network, and execution units, etc. to estimate dynamic power dissipation in a CPU. Other models (SimplePower [96], TEM2P2EST [27]) employ empirical models based on known circuit implementations for better accuracy. These models trade-off ease and speed of simulation for estimation accuracy and/or scalability across process technologies. In addition, these models do not accurately model power dissipation for wires and interconnects. Given the unique design issues in DSAs, the market need for extremely tight design schedules, and the lack of accurate but flexible power models, the CoGenE simulator attempts to address the above issues.

In CMOS circuits, the dynamic power dissipation (P_d) is defined as

$$P_d = aCV^2f,$$

where f and V are the frequency and voltage of operation of the circuit, respectively, C is the load capacitance, and a is the switching activity factor. The performance model estimates the activity factor for structures such as FIFOs, buses, caches, etc. For internal circuits, where the modeling is not accurate enough to calculate activity factor, we assume toggle rate values, as recommended by the Wattch model [14, 6]. Leakage power, which also contributes to a significant fraction of total power dissipated in CMOS circuits today, is determined using analytical models for memory structures (similar to HotLeakage [107]) and empirical table lookup models for all other circuits.

6.1.1 Analytical Models

These types of models are employed for parameterizable regular structures and we employ a methodology similar to Wattch [14] to build the models for various structures. Each of the structures are broken into different constituent stages and equivalent RC models are built for each of them. Finally, we add the capacitances for each of the stages and then calculate the dynamic power for the structure. This type of modeling is relatively well understood and models power for wires internal to a circuit as well. Table 6.1 shows the type of modeling available for each type of structure within a DSA. For structures represented by both the models, the choice of model is dependent on the required level of accuracy, speed of simulation, and the required level of detail.

6.1.2 RTL-based Empirical Models for Dynamic and Leakage Power

This type of modeling is employed for all structures where the underlying implementation varies across different units and in structures where it is difficult to build parameterizable analytical models such as control circuits, custom data-path, arithmetic units, etc. Power dissipation for such structures is determined by the activity factor of data and the control signal that determines the type of operation performed in the structure. For example, in an FIFO circuit, the control signal (push, pop) determines the operation type and the activity factor of input data determines the switching activity in the circuit. Hence, power dissipation is computed for various activity and control values using commercial low-level power simulators similar to Ramani [81] and form a table for the circuit. The table contains both the dynamic and the leakage power for the circuit. For power estimation, we perform a table lookup based on the control signals and activity. Table 6.2 shows an example for a FIFO that is modeled empirically.

Table 6.1. Types of models available for the different structures within a DSA

Structure	Analytical Model	Empirical Model
Cache	dynamic, leakage	dynamic, leakage
FIFO	dynamic, leakage	dynamic, leakage
Register file	dynamic, leakage	dynamic, leakage
Bus	dynamic, leakage	-
Crossbar	dynamic, leakage	dynamic, leakage
Arbiter	dynamic, leakage	dynamic, leakage
HLU	-	dynamic, leakage
AGU	-	dynamic, leakage
Arithmetic	-	dynamic, leakage
data path	-	dynamic, leakage

Table 6.2. Empirical Table for a FIFO

Activity Factor	Control (push, pop)	Dynamic Power	Leakage Power
0.2	00	0.105	0.118
1.0	00	0.105	0.118
0.5	01	1.610	0.122
.	.	.	.
.	.	.	.
0.7	11	1.610	0.122

6.1.3 Interconnect Power Models

Power dissipation on interconnects contributes to a major fraction of the total chip power [56]. An analytical model of a bus is used to estimate power dissipation for global and local buses that contribute a significant fraction to the total power of a macroblock. For global buses, a methodology similar to [5] is used, with appropriately sized buffers and repeaters, interbuffer distances, etc., depending on the delay and power requirements. For other interconnects, we employ analytical models similar to [100] for matrix-based crossbars, arbiters, and empirical models for a multiplexer-based crossbar.

6.2 Evaluation Methodology

CoGenE compiled code running on the DSA is also compared to three other design options, all of which were normalized [91] to a 0.13μ process :

1. Software running on a 400 MHz Intel XScale processor that represents a highly energy efficient embedded processor. The Xscale does not have floating point instructions, and so, we make our comparisons against an idealized Xscale, where all floating point operations are replaced by integer operations. The code is then run on an actual Xscale processor and performance and power consumption are measured.

2. Software running on a 2.4 GHz Intel Pentium 4 processor that can support the real time requirements of the face recognition kernels.

3. Manually scheduled microcode implementation running on the simulated cluster architecture representing the best performance point. Energy and performance numbers are calculated using Synopsis Nanosim, a commercially designed spice level simulator, on a fully synthesized and back-annotated Verilog and Module Compiler-based implementation. The results are then normalized to a 0.13μ process. Normalization was done by employing conservative constant field scaling [91]. The simulated model includes a full clock tree and worst-case wire loads based on assigning wire parasitics based on metal 1. Hence, these results are pessimal since in a fabricated design, the long wires would be routed on larger metal layers.

6.2.1 Benchmarks

Our benchmarks consists of seven kernels from face recognition, three kernels from speech recognition, and six kernels from wireless telephony domains. The face recognition kernels constitute the different components in a complete face recognition application. To

increase the robustness of the study, we employ two fundamentally different face recognition algorithms. The EBGM algorithm is more computationally intensive compared to the PCA/LDA recognition scheme. All the face recognition kernels were obtained from the CSU face recognition suite [25]. The speech recognition application consists of three phases that contribute to 99% of total execution time: preprocessing, HMM, and the Gaussian phase [61]. The kernels from the wireless domain include predominant operations like matrix multiplication, dot product evaluation, determining maximum element in a vector, decoding operations like rake and turbo, and the FIR application. Finally, we employ three kernels from the ray tracing domain. A description of the benchmarks are provided in Table 6.3.

6.2.2 Evaluation Metrics

To effectively compare the performance of different architectures, we employ throughput measured in terms of the number of input frames processed per second. We employ the energy-delay product as advocated by Horowitz [33] product to compare the efficiency of different processors since both energy and delay for a given unit of work are conflicting constraints for the architect and circuit designer. We employ pruning ability and exploration time as metrics to evaluate the efficiency of design space exploration. Given the complete design space, degree of pruning gives us a measure of the reduction in the size of the exploration space. The total time for exploration evaluates the time taken to arrive at optimal design points for various constraints.

Table 6.3. Benchmarks and Description

Benchmarks	Description
Face Recognition	
Flesh Toning	preprocessing for identifying skin toned pixels
Erode	First phase in image segmentation
Dilate	Second phase in image segmentation
Viola Detection	Identifies image location likely to contain a face
Eye Location	Process of locating eye pixels in a face
EBGM recognition	Graph based computationally intensive matching
PCA/LDA recognition	Holistic face matching
Speech Recognition	
Preprocessing	Normalization for further processing
HMM	Hidden Markov Model for searching the language space
GAU	Gaussian probability estimation for acoustic model evaluation
Wireless Telephony	
Vecmax	Maximum of a 128 element vector
matmult	Matrix multiplication operation (integer)
dotp_square	Square of the dot product of two vectors
Rake	Receiving process in a wireless communication system
Turbo	decoding received encoded vectors
FIR	Finite Impulse response filtering
Ray Tracing	
Traversal	Ray intersection with acceleration structure
Intersection	Ray intersection with primitive objects
Shading	Computing color and illumination of pixel

CHAPTER 7

SCA DESIGN EXPLORER

DSA design space exploration (DSE) involves a number of choices in each of the three subsystems: memory, interconnect, function units. The simplest choice set is the function unit subsystem given that the choice is at a high grain-level of integer, floating point unit, or register file. Width and the number of registers and ports are also choices. The interconnect layer is composed of one or more multiplexer layers and each multiplexer has a choice of widths. The memory subsystem is a bit more complex. SRAM choices involve width, size, and number of ports. AGU's perform affine address computations but can vary in number. The HLU can have one or more contexts. A summary of the current design space choice options and costs are summarized in Table 7.1. Dilation and thinning is obvious for all but the interconnect subsystem where dilation means adding levels or widening one or more multiplexers. Thinning reverses this choice. Increasing the number of levels increases the fall through delay but may improve frequency, while widening a multiplexer increases the delay of that component and may reduce frequency. Given the number of design choices and the number of kernels within the application, the combined set of options may create a design space that is too large to exhaustively examine. In order to simplify the process, only one architectural feature is changed per iteration and the choice is based on the lowest cost. Making too many changes can lead to a feedback loop where the exploration algorithm gets stuck in a local minima. Then the choice is which subsystem to change first. During the course of this work, it was found that since the biggest performance problem typically lies in function unit starvation. We therefore choose to modify the memory system first, then the functional units, and then the interconnect in order to balance the function unit requirements with memory system capability. The process iterates to address additional imbalances across the subsystems.

Table 7.1. Design space and cost for each functional unit variable

Component	Range	Performance cost	Energy cost	Compiler cost	Total cost
Data width	16, 32, 64 (bit)	1	1	1	3
SRAMs (input, output, and scratch)	1, 2, 4, 8, 16, 32, 64 (KB) each	0	0	0	0
Ports (SRAMs and RF)	1, 2, 3 each	1	1	1	3
Hardware loop unit contexts	1, 2, 3, 4, 5	0	1	0	1
AGUs	1-8 (increments of one) per SRAM	0	0	1	1
Register file size	8, 16, 32 entries	1	0	0	1
Register file number	1,2,3,4,5,6	1	1	0	2
Functional unit type	integer, floating point	-	-	-	-
Functional unit mix	multiplier, adder, compare, etc.	-	-	-	-
Functional unit number	1-8	0	1	0	1
Interconnect Width	2-5	0	1	1	2
Interconnect levels	1-3	1	1	1	3

7.1 DSE Using Stall Cycle Analysis (SCA)

SCA is a simple idea; namely, whenever the compiler's instruction schedule is delayed due to resource contention or whenever stalls occur in simulation, then there must be a bottleneck culprit. These culprit points are logged, classified by culprit type, and quantified in terms of their impact. Examples of such logged statistics are average functional unit utilization rate, register file utilization rate, contention rate in the interconnect subsystem, execution time, energy dissipation, etc. The major overheads that are detrimental to performance or energy are:

- Function unit starvation is due to the inability of the memory system to deliver data to the function units at the right time. The culprit may be too few AGUs, not enough HLU contexts, interconnect contention, SRAM port contention, or insufficient SRAM capacity indicated by a high SRAM miss rate.

- Insufficient hardware to support the available application parallelism. This bottleneck arises when there are more independent instructions than can be issued in a cycle. High function unit or interconnect contention helps identify the culprit.

- Under utilized function units may be caused by starvation or by having more than are needed.

- Routability problems will force values to be stored in either pipelined or centralized register files. High interconnect path contention identifies the interconnect culprit and can be fixed by widening multiplexers or increasing the number of multiplexer levels.

7.2 Associating Cost for Architectural Attributes

Culprit solutions vary with culprit type and importance and the cost of each dilation or thinning option guides the choice. In order to pick the best solution, a predetermined cost is associated with each resource choice. Resource cost is based on the improvements in performance, energy, or compilation complexity that the resource provides. If the increase in size or number of a particular unit delivers a significant increase in performance, then the unit is designated with a low cost for performance, and vice versa. For energy or code generation complexity, a high cost is assigned if the unit significantly increases energy or compilation complexity. In this dissertation, a simplified Boolean cost model is used: 1 for high and 0 for low. In general, the system user may assign costs as any integer or floating point value. The total cost is the weighted sum of the performance, energy, and code generation costs. The SCA approach defines the best solution to be the one with the lowest cost.

Although assigning various weights to the three metrics can lead to interesting search spaces, CoGenE restricts itself to equal weights in this study. The notion of assigning cost is nontrivial in certain cases. For example, applications with multiple loop contexts benefit significantly in performance when an HLU is present. The HLU [62] automatically updates the loop indices for all the loop contexts and generates indices for the AGU to perform address calculations. The HLU contains its own stack, registers, and adder units. The addition of a HLU has the potential to deliver very high performance, but energy dissipation increases. It also provides hardware support for modulo scheduling of loops whose indices are not known at compile time. This reduces code generation complexity. Hence, the unit is assigned a low performance cost, high energy cost, and a low code generation cost. Table 7.1 shows the costs for each of the architectural resources.

7.3 Design Selection

The importance of choosing the best initial design choice is significantly reduced given that the design space will be automatically explored. A poor choice will result in more iterations but the results of the process will be very similar. Hence, the starting point for design selection (DSEL) is: a 1 KB single ported input, scratch, and output SRAM; one AGU per SRAM, a single context HLU, one floating point unit, one integer unit, and a single level interconnect using 4-wide multiplexers. For this configuration, high initial SRAM miss rates cause back-end starvation. Although different solutions in the memory subsystem (adding HLU contexts, AGUs, increasing ports) can be employed, the low-cost

solution is to increase the size of the input and output SRAMs. After an arbitrary number of iterations, a further increase in SRAM size may provide minimal performance improvements and an incommensurate energy increase. The choice then is to reduce their sizes for greater energy savings at minimal/no performance loss. SCA proceeds with the addition of AGUs before an increase in HLU contexts is chosen. Increasing the number of contexts arbitrarily increases the area, complexity, and power dissipation while providing minimal performance improvements. The key to HLU configuration choice is to provide what the AGU's need, hence AGU dilation precedes HLU dilation.

Since the initial design choice only has one floating point and one integer unit, high function unit contention will be observed for any nontrivial application suite. Function units with high utilization and contention are dilated by one for each type as needed. An increase in the number of functional units entails an increase in the length of the instruction word, which increases power dissipation in the instruction cache and interconnect. The register file use rate and interconnect utilization metrics are employed to increase/decrease the size and number of register files. Routability problems implies increasing multiplexer width. If a frequency target is specified, then multiplexer width will be constrained and an additional interconnect level and the associated pipeline register will need to be investigated. This increases compilation complexity and may lead to infeasible schedules. For this case, the algorithm returns to the previous design point.

7.4 SCA Exploration Algorithm

In summary, the DSE steps are:

1. Collect program statistics during compilation and simulation.
2. If function unit starvation is evident, then optimize the memory subsystem. Modify the architecture description file and go to step 1.
3. If function unit contention is seen, then optimize the function unit selection. Modify the architecture description file and go to step 1.
4. IF high interconnect contention is observed, then optimize the interconnect subsystem. Modify the architecture description file and go to step 1.

The process iterates until a set of near-optimal designs are found. In cases where the algorithm cannot provide a feasible design, the tool returns to the last iteration.

CHAPTER 8

EVALUATION

The design goal of the instruction scheduling algorithm is to provide real-time performance with minimum energy. In order to evaluate the throughput and energy control capabilities of CoGenE, CoGenE is compared against the performance of hand scheduled code on a Pentium 4 (Figure 8.1). The result is then compared to an XScale-based implementation for energy consumption. Finally, a comparison of the two face recognition algorithms is presented.

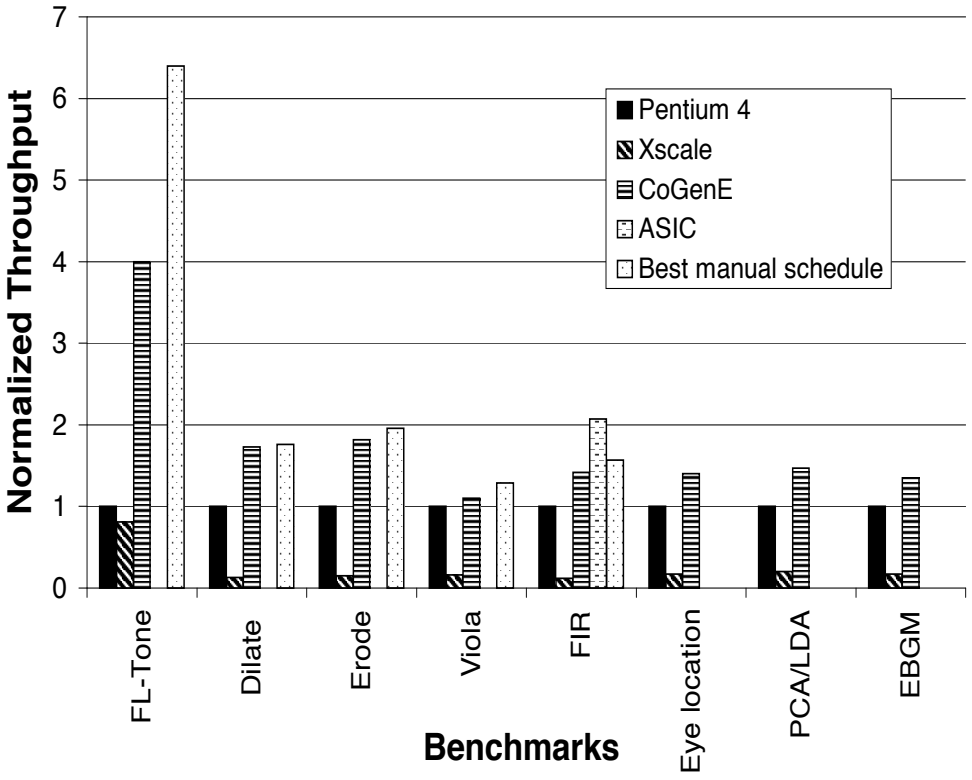

Figure 8.1. Throughput comparisons for different configurations

8.1 Face Recognition Evaluation

Figure 8.1 compares the throughput (number of input frames processed per second) for the different processors. The hand coded implementation delivers the best throughput. The CoGenE version delivers a throughput that is 1.65 times better than the Pentium 4 processor and 8.64 times better than the XScale processor. This underlines the fact that our CoGenE framework exploits the streaming nature of the face recognition kernels to deliver the throughput necessary to achieve real-time constraints. CoGenE is able to achieve 85% of the throughput of manually scheduled code. Figures 8.2 and 8.3 show the energy consumption per input and the energy-delay product comparison for the different processors. The CoGenE compiler reduces energy consumption by 9.25x when compared to the low-power XScale processor. The energy advantage comes from efficient decoupling between address and data computations provided by the loop unit and AGUs, and by minimizing communication overhead due to the ASIC-like pipeline structures. The result is a DSA that performs face recognition at embedded energy budgets. It is noteworthy the energy-delay product of the Xscale processor is within 35% of the Pentium 4 processor, and that our approach provides 80x improvement over the Xscale. The improvements are

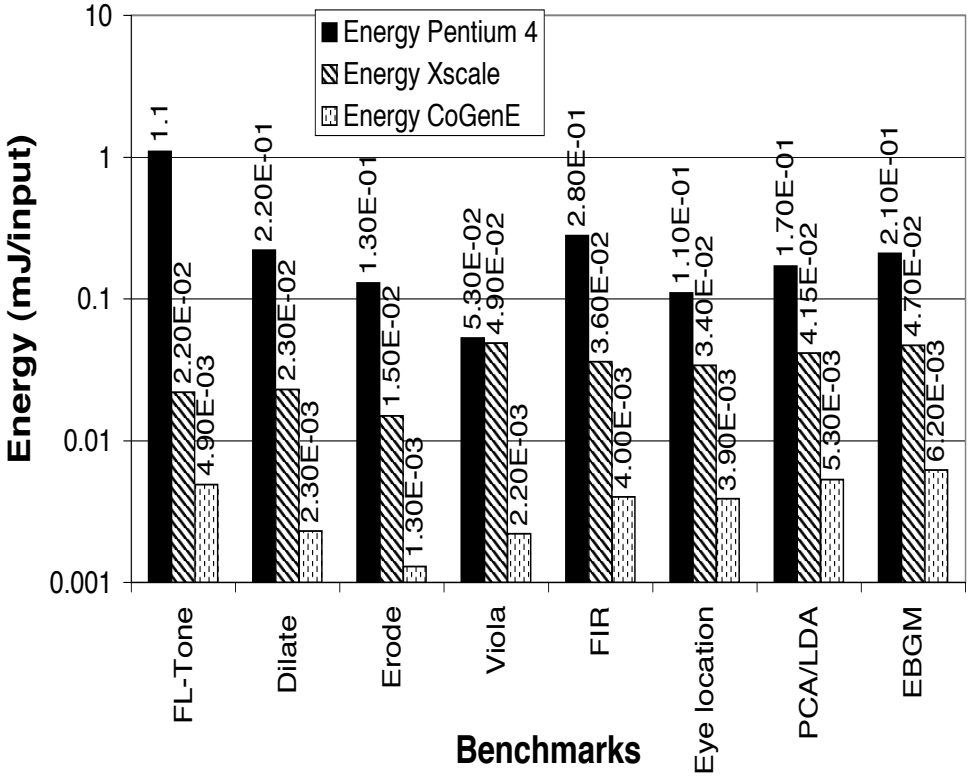

Figure 8.2. Energy/input packet comparison

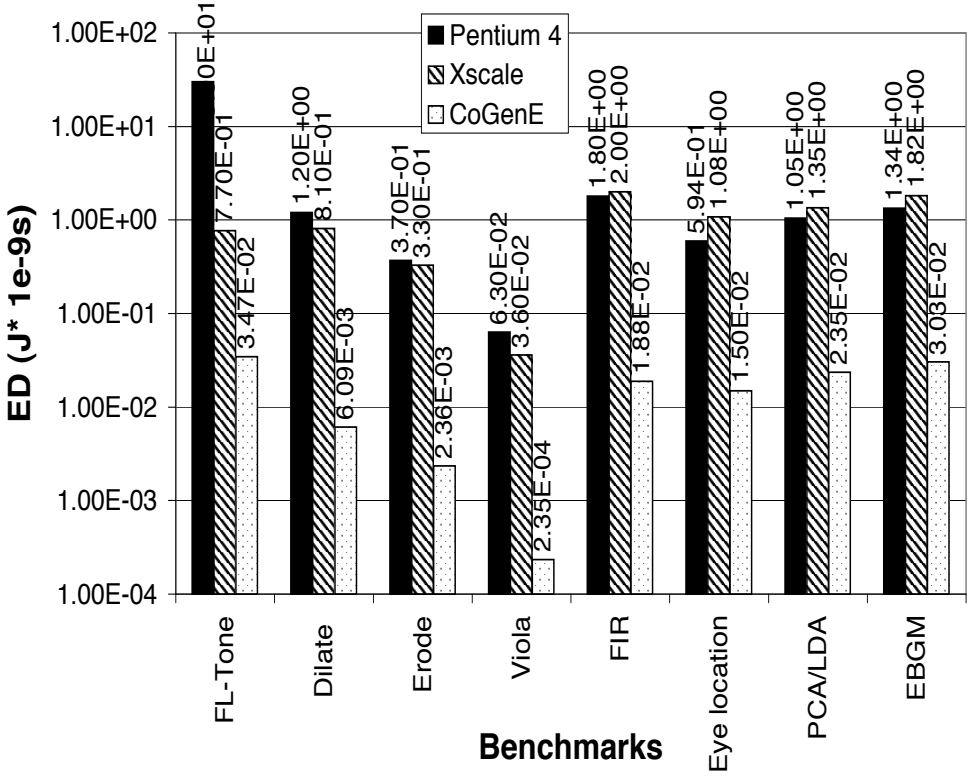

Figure 8.3. Energy-delay product comparison

consistent across all the applications in the domain. It is interesting that flesh toning accounts for less than 5% of the total execution time but consumes an incommensurate amount of the total energy. This is because the floating point parallelism in flesh toning exceeds the number of floating point units (four) available in the cluster. This means intermediate results must be saved and retrieved from the register file, which is inefficient. The hand scheduled code does a better job of vectorizing the code, which indicates that further scheduling improvements are possible. CoGenE does well on the image segmentation phase (erode and dilate kernels) , and the architecture delivers two orders of magnitude better energy-delay product than the XScale.

The Viola/Jones face detection algorithm is characterized by a recurrence that involves two adjacent image rows and an additional row for intermediate for intermediate storage. The algorithm sweeps over the image by operating on a 24x24 window. The algorithm then successively shifts by one pixel position. Pixel value lifetimes are therefore high. The architecture benefits as a result and reduces energy consumption by as much as 22x over the XScale.

The CoGenE FIR version delivers two orders of magnitude energy-delay product improvement over the XScale processor and is only 24x worse than the ASIC implementation.

This is partly because the ASIC possesses significantly more functional units than our architecture.

8.1.1 PCA/LDA vs EBGM

One of the goals of this study is to compare two fundamentally different face recognition algorithms and to identify the algorithm that is better suited for hardware implementation. The PCA/LDA algorithm is a holistic image comparison algorithm as opposed to the EBGM algorithm. The EBGM algorithm requires an additional normalization step after face detection to increase the accuracy of the algorithm. This adds computational complexity in the algorithm and contributes to the 9% performance advantage of PCA/LDA algorithm. The PCA-LDA algorithm also has a 17% advantage in energy and a 30% advantage in energy-delay product. We then reduced the number of facial feature nodes in the EBGM algorithm in order to reduce complexity but found that accuracy immediately fell to unacceptable levels. The conclusion is that the PCA-LDA algorithm is superior for our architecture and compilation approach.

8.2 SCA Results

Embedded designers typically attempt to design a DSA to meet a given performance and energy budget (Figure 8.4) and then optimize the area for the design. SCA is employed in a similar manner and attempts to search through the design space for a set of designs that meet the minimum performance and energy budgets. In the first case study, the impact of SCA is evaluated in designing an optimal domain-specific architecture for the face recognition domain. The seven benchmarks required for face recognition are fed as inputs to the framework for iterative exploration. We then discuss the design of optimal DSAs for speech recognition and wireless telephony. Each of the energy-Delay optimal DSAs are compared to the best manual designs from previous studies and also to industrial design points (wherever applicable) for performance and energy dissipation.

8.2.1 DSA for Embedded Face Recognition

Figure 8.4 shows the SCA design points for the seven kernels in the face recognition suite. It shows the throughput and energy dissipation for each of the design points, starting with the usual initial design point (1 INT, 1 FPU, 1 KB input, scratch, and output SRAMs, 1 AGU/SRAM, no HLU) and observe that its throughput is about fives times slower than real-time performance set at 5 frames/sec.

Minimum real-time performance is shown by a vertical line normalized to 1. All points

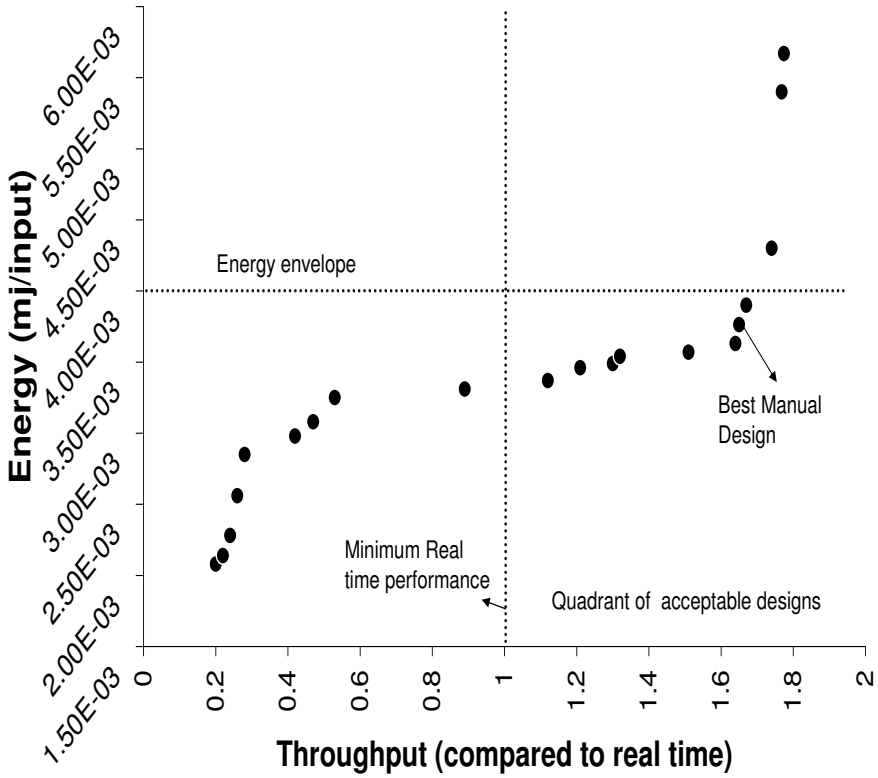

Figure 8.4. SCA applied to face recognition

to the left of the line do not meet performance goal. SCA successively increases the size of the input and output SRAMS to 8 KB with up to 2 AGUs/SRAM. At this point, throughput starts to saturate and this is indicated by a low SRAM miss rate and very high utilization of AGU and interconnect resources. SCA then adds an HLU and successively increases the number of contexts to improve performance. Once memory optimization is complete, SCA dilates function unit resources and significant increases in performance is observed. A configuration of 3 INT + 3 FPU function units achieves the minimum required performance. All design points to the right of this configuration are checked against the energy requirements. The horizontal line indicates the energy budget and was set to be one order of magnitude better than the XScale. The feasible design quadrant contains designs that meet both energy and performance constraints and the user can then choose a particular design for fabrication.

Our previous best manual design [61, 77] comprised: three 8 KB SRAMs, with 3 HLU contexts, and a 8 way VLIW (3 INT + 4 FPU + 1 register file)) and was shown to be 1.65 times faster than the minimum required real-time performance with a 10x energy benefit when compared to the XScale. Exploration also found this design point (Figure 8.5). The energy-delay plots demonstrate that the architecture was designed for close to optimal

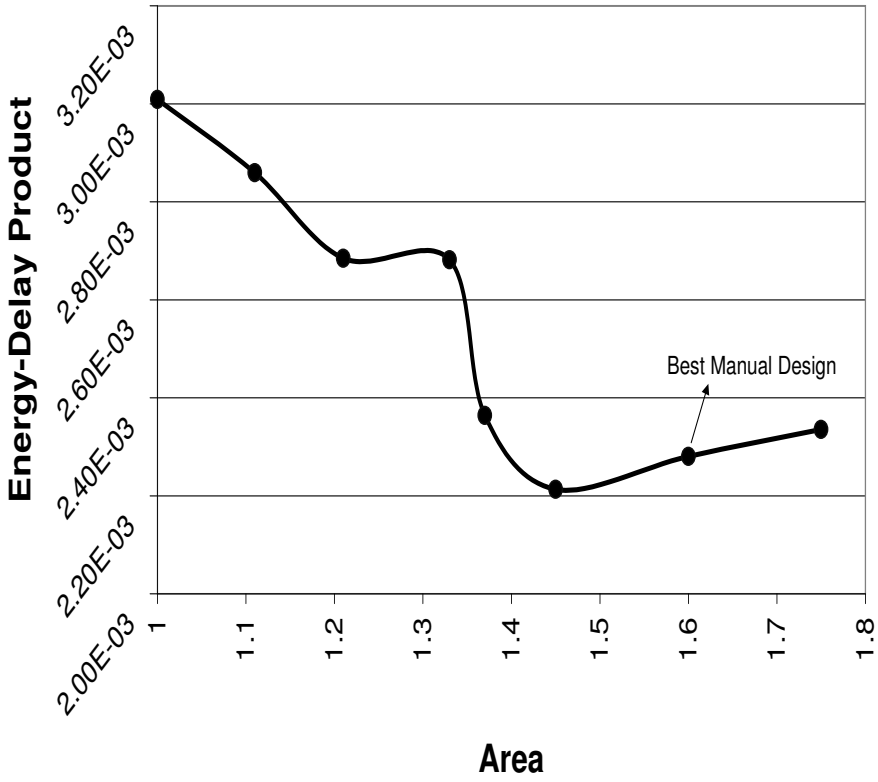

Figure 8.5. Energy-delay product comparisons for performance-energy designs

energy-delay characteristics. Table 8.1 shows that SCA also found a configuration with a 4KB scratch SRAM and an additional integer unit to have a 4% energy improvement and a marginal energy-delay product improvement over the manual design. Manual design is error prone and extremely time consuming. This case study indicates that similar or better results can be found rapidly by exploring additional design points. Due to rapid SCA, this exploration investigated fewer than forty design points in a design space of approximately 1000 points. The total exploration time was 215 minutes on a 1.6 GHz AMD Athlon PC.

For design points in the "acceptable" quadrant, Figures 8.6 and 8.7 show the energy-delay product, energy, and throughput with respect to area. The user can choose the appropriate design and example choices could based upon:

- *Minimum area*: a 6-way function unit design barely meets the performance requirement and occupies minimum area. It is approximately 75% smaller than the design with highest performance.

- *Minimum energy-delay product* : a 9-function unit design delivers the best energy-delay product and is marginally better than the manually designed system.

Table 8.1. Best configurations for different constraints, throughput, and energy comparisons for different targets

Design Point	Memory (KB)	Scratch (KB)	AGUs	HLU con.	INT units	FP units	RF	Mux levels	Throughput	Energy (mj/inp.)
Face Recognition										
Manual	8	8	2x3	3	3	4	yes	one	1.65	3.76e-03
Min. Area	8	1	2x3	3	3	3	no	one	1.12	3.46e-03
Min. EDP	8	4	2x3	3	4	4	yes	one	1.64	3.63e-03
Max. Perf.	8	8	2x3	3	5	5	yes	one	1.68	3.80e-03
No HLU	16	16	2x3	-	8	5	yes(2)	one	1.14	5.4e-03
Speech Recognition										
Manual	8	8	2x3	2	4	4	no	one	1.98	1.1e-03
Min. Area	8	2	2x3	2	3	2	yes	one	1.03	0.76e-03
Min. EDP	8	8	2x3	2	4	3	yes	one	1.92	1.13e-03
Max. Perf.	16	16	2x3	2	5	5	yes	one	2.74	3.7e-03
Wireless Telephony										
Manual	4,2	2	4	-	12	-	yes (4)	two	-	100x (EDP)
Min. Area	2	1	3	-	8	-	yes (1)	one	-	50x (EDP)
Min. EDP	4	8	4	-	10	-	yes (3)	one	-	120x (EDP)
Max. Perf.	16	8	6	-	16	-	yes (4)	three	-	30x (EDP)
No clusters	16	16	8	-	15	-	yes (5)	one	-	17x (EDP)
All three domains										
Min. ED product	8	8	6	3	8	4	yes (1)	two	-	-

- *Maximum performance*: An 11-function unit design is approximately 50% faster than the design with minimum area.

8.2.2 DSA for Embedded Speech Recognition

Table 8.1 shows the DSA configurations for all three case studies in terms of minimum area, minimum energy-delay product, and maximum performance for the feasible designs. For speech recognition, the configuration with minimum area reduces energy dissipation by 44% when compared to the best manual design [61]. Similarly, the configuration with the highest performance delivers a performance improvement of 38%. The manual design was optimized for energy-delay product and the SCA design is only 5% worse. The primary cause is that the manual design used wider multiplexers than was possible for the SCA approach. The SCA width limit was set with a particular frequency limit in mind and therefore, SCA did not explore the manual design point. The width limit could easily be changed but even with a more restricted component space, the SCA result is close and did not require a man-year to find. Total exploration time was 183 minutes.

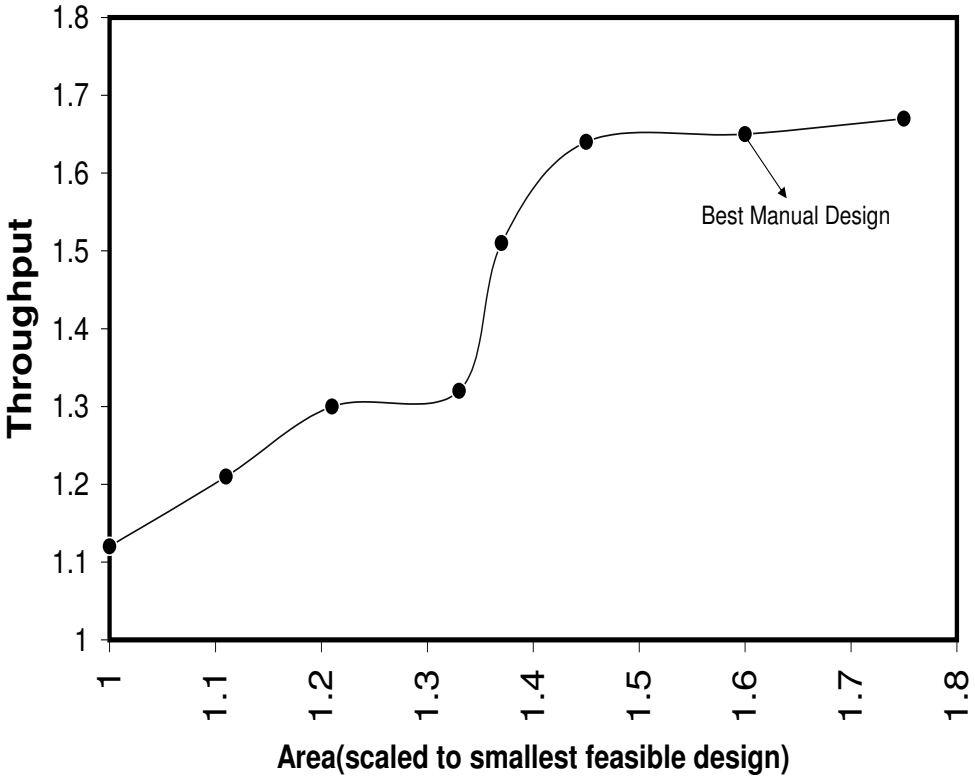

Figure 8.6. Throughput comparison for performance-energy designs

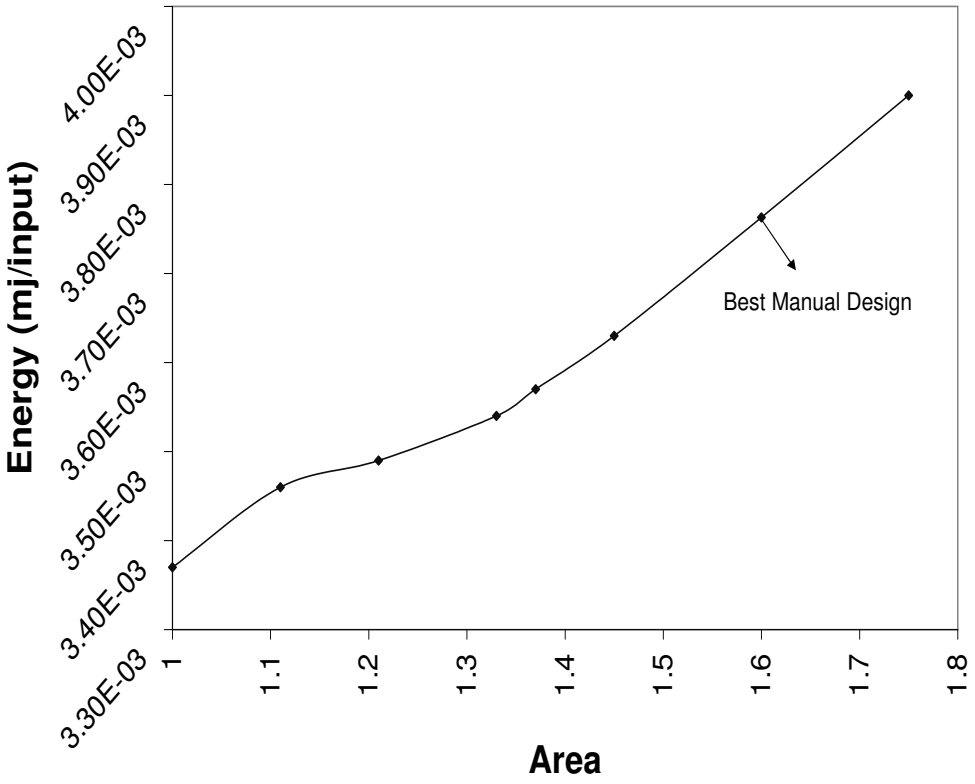

Figure 8.7. Energy comparisons for performance-energy designs

8.2.3 DSA for Wireless Telephony

The architecture for wireless telephony is significantly different from those for the recognition algorithms. The basic data-path and function unit width is 16-bits rather than the 32-bit paths found in speech and face recognition. SCA, therefore, thins data paths from the default 32-bit initial design point. Investigating HLU addition, SCA finds that no performance gain results. The domain uses a large number of constants that are regularly accessed. This results in the need for multiple register files and a two level interconnect. Each register file supports a few integer units and while SCA does not perform clustering directly, the introduction of a second level interconnect across the functional units provides this in an indirect fashion.

The design configuration with minimum area is a single cluster machine that barely meets the performance requirements. The design is balanced in terms of data throughput across memory, interconnect, and function units, but there is more available parallelism in the application space. Dilation in SRAM size and function units could extract the parallelism. SCA employs dilation and selects a configuration that equals the performance of the manual design [43] but at a lower energy dissipation. This design provides a 17% improvement over the manual design. Further dilation increases pressure on the interconnect and SCA observes diminishing returns in performance. SCA dilates interconnect width until the frequency limits are met. Beyond that, the introduction of a multilevel interconnect facilitates clustering and allows dilation in the functional units. The design with maximum performance consists of a three-level interconnect and supports as many as sixteen integer units and four register files. Clustering increases the search space and makes it difficult to manually identify the most optimal designs. This makes a case for tools that explore the design space automatically.

8.2.4 Impact of Per Design Code Generation

As opposed to previous studies [49] that do not consider the impact of micro-architectural changes on code generation, our study generates optimized code for each of the kernels in the suite and hence, guarantees compilation for every design candidate. The final set of acceptable designs represent a synergy between compilation and architectural design. To evaluate the benefit/demerit of per design code generation, there are two interesting scenarios for which the presence of a particular functional unit delivers significant performance and energy improvements.

In the first scenario, the face recognition case study is evaluated where compiler support is disabled for an HLU. In the absence of a HLU, SCA moves to optimize the function

units and successively increases the number of integer units and the size of the register file. After performance saturation, it optimizes the memory subsystem and increases the size of the SRAMs and observes a performance improvement. Due to the absence of the HLU, the optimization algorithm successively increases memory size and integer units to account for loop computations, storage, and additional interconnect bandwidth. The extra computations also introduce the problem of port saturation. The result is a completely different set of feasible candidates. The candidate with the best energy-delay product (shown as No HLU in Table 8.1) meets the performance budget, but is 30% slower while dissipating 35% more energy.

In the second scenario, the number of interconnect levels in the framework is limited for the wireless telephony domain. SCA identifies a different configuration that delivers the performance and the energy requirements for the domain. Nevertheless, this configuration degrades energy-delay product by 44.3% with respect to the unrestricted design with the best energy-delay product. Multiple interconnect levels reduce congestion for data at the interconnect and deliver both performance and energy advantages for this particular domain. Note that imposing this limitation on the other two domains does not affect the search results since their critical bottleneck is not in the interconnect subsystem.

8.2.5 Sensitivity Analysis: SCA Robustness

An interesting option is to evaluate the robustness of SCA in designing a single DSA for all 3 domains. While convergence was slow, SCA arrived at an energy-delay optimal design comprising 12-way function units (8 INT + 4 FPU) with a centralized register file supported by a well-provisioned memory system (8 KB input, output and scratch SRAMS), 6 AGUs, an HLU with three contexts, and a two level interconnect). SCA pruning was effective investigating approximately one hundred design points in a design space of over 3000 design points. Total exploration time was split into 143 minutes for speech recognition, 187 minutes for face recognition, and 85 minutes for wireless telephony phases. When compared to an architecture for one domain, this design consumes 80% more energy than the sum of 3 separate DSAs, but is capable of delivering the real-time performance for all three applications in less area. Manually examining such a large design space would be intractable.

A good exploration algorithm should not be aliased significantly by the initial design choice. In order to test this aspect, two different starting points were chosen: one that exceeded the energy envelope required for embedded applications, and another point that is in the middle of the feasible space. For the first test point, SCA performs exploration

by successively removing architectural resources and converged to the feasible space in less than 20 iterations. For the second test point, SCA discovered all the feasible design points by successive addition of resources in less than 40 design points. In both cases, convergence is sufficiently fast and delivered the same designs for the face recognition and the wireless telephony domains. In the case of speech recognition, the tests lead to similar but not exactly the same configurations. The difference was in the size of the register file and this delivered a marginal energy-delay product difference of less than 2%. This was caused by slightly different instruction schedules for the two candidates. The conclusion is that our SCA method results are reasonably independent of the initial design point choice.

CHAPTER 9

RAY TRACING

The wide availability of commodity graphics processors has made real-time graphics an intrinsic component of the human/computer interface. These graphics cores accelerate the z-buffer algorithm and provide a highly interactive experience at a relatively low cost. However, many applications in entertainment, science, and industry require high-quality lighting effects such as accurate shadows and reflections. These effects are difficult to achieve with z-buffer algorithms, but are much easier to achieve using ray tracing. Although ray tracing is computationally more complex, the algorithm exhibits better scaling properties than the z-buffer approach. Nevertheless, ray tracing memory access patterns are difficult to predict and therefore, the parallelism speedup promise is hard to achieve.

CoGenE has evolved from the study of recognition and cellular telephony domains. Designing a DSA for ray tracing using the same approach serves as a stress test for the approach. To efficiently accelerate ray tracing with CoGenE, the native recursive algorithm is transformed into a stream filtering problem. While stream-based processing is well-suited to recognition and ray tracing, some high-level differences emerged between the domains. The VLIW approach is a good match for recognition applications but the SIMD approach suits ray tracing. In addition, ray tracing also requires hardware support for scatter/gather operations to accelerate the complex memory access patterns. This dissertation highlights stream filtering, a novel software approach to ray tracing, and proposes StreamRay, a wide-SIMD multicore architecture that delivers high performance for ray tracing. CoGenE is employed in synthesizing the execution and the interconnect subsystem. In addition, the parallelism benefits of the DSA approach are employed in designing StreamRay.

9.1 Importance of Ray Tracing

The *visibility problem* is a fundamental problem in computer graphics applications: given a set of three-dimensional (3D) objects and a viewing specification, the task is to determine which lines or surfaces are visible from that view point. Currently, the z-buffer algorithm

and ray tracing are the two most prevalent approaches used to solve the visibility problem. Graphics processing units (GPUs) have significantly enhanced human/computer interfaces by accelerating the z-buffer algorithm [18], which in its most basic form consists of a loop over the objects in a scene:

```
foreach object N do
    foreach pixel P through which N might be visible do
        if z_new < z_pixel then
            c_pixel = c_new
            z_pixel = z_new
        end
    end
end
```

The z-buffer algorithm projects an object toward the screen and updates the corresponding color and distance (or z) values, but only if the new z value is less than the current z value associated with the pixel. While hardware implementations of this algorithm provide highly interactive environments for many computer graphics tasks, it is not well-suited for applications that require high-quality visual effects such as shadows, reflection, and refraction. In contrast, the basic ray tracing algorithm [101] consists of a loop over all of the pixels in an image:

```
foreach pixel P do
    foreach generated ray R do
        find nearest object visible through P by R
        update c_pixel
    end
end
```

A 3D line query is used to find the nearest object in the parametric space of the ray. Typical implementations of the algorithm employ a hierarchical data structure to quickly eliminate large parts of the search space and accelerate the query, thereby leading to improved performance.

Ray tracing boasts several key advantages over the z-buffer algorithm. First, for preprocessed models, ray tracing is sublinear in the number of objects, N; thus, for some sufficiently large value of N, ray tracing will always be faster than the z-buffer algorithm, which is linear in N [22]. Second, the computational kernel of the algorithm performs a 3D line query, and that same operation can be reused to generate global illumination effects

such as shadows, reflection, and refraction [101]. Thus, the same operation that is used to solve the visibility problem can be used to render high-quality visual effects as well. Third, ray tracing is highly parallel and has been demonstrated to have over 91% efficiency on 512 processors [71]. This characteristic, combined with the advent of multicore microprocessors and algorithmic developments, make ray tracing an attractive alternative for interactive rendering if a solution can be found to mitigate problems associated with ray tracing's less predictable memory access patterns.

9.2 Stream Filtering for Coherent Ray Tracing

Coherent or packet-based ray tracing [99] enables the efficient use of SIMD processing. In this approach, rays are processed in coherent groups utilizing SIMD instructions such as SSE or Altivec. However, when rays begin to diverge, a large percentage of the packet's rays do not actively participate in the same computations. As a result, unnecessary work is performed on what is called the *inactive* subset. The result is decreased performance and increased power consumption.

The stream filtering approach recasts the basic ray tracing algorithm as a series of filter operations that exploit coherence by partitioning arbitrarily sized groups of rays into active and inactive subsets. Initial work [36] has shown that streams of sufficient length exist in all stages of ray tracing to make wide SIMD processing an attractive alternative to packet-based methods. This approach is based on two core concepts: (1) *streams* of rays, and (2) sets of *filters* that extract substreams with certain properties. Both are usefully applied to the major stages of ray tracing: traversal, intersection, and shading.

9.2.1 Core Concepts

A *ray stream* contains data of the same type and can be of arbitrary length. A *stream filter* is a set of conditional statements on the elements of a ray stream:

```
out_stream filter<test>(in_stream)
{
  foreach e in in_stream
    if (test(e) == true)
      out_stream.push(e)
  return out_stream
}
```

The core operations in ray tracing, including traversal, intersection, and shading, can be written as a sequence of conditional statements that are applied to each ray [36]. With stream filtering, instead of applying conditional statements to individual rays, the state-

ments are executed in SIMD fashion across groups of N rays to isolate those rays exhibiting some property of interest. In an N-wide SIMD environment, filters are implemented as a two-step process: conditional statements are first applied to groups of N elements from the input stream, generating a Boolean mask. To create the output stream, the input stream is then partitioned into active and inactive subsets based on that mask:

```
out_stream filter<test>(in_stream)
{
  foreach simd in in_stream
    mask[simd] = test(simd)
  out_stream = partition(in_stream, mask)
  return out_stream
}
```

Nonsequential memory access patterns require scatter/gather operations to generate a sequential stream of ray data from the stream elements. Thus, one important performance requirement for the stream filtering approach is hardware scatter/gather support.

9.2.2 Coherence

Wide SIMD units can be used to process arbitrarily sized groups of rays with high efficiency for two reasons: first, the algorithm exploits parallelism when processing streams as a sequence of groups with N elements; second, stream filtering removes any elements that would perform unnecessary work in subsequent stages of the rendering process, allowing these stages to process only active elements. In fact, the output stream created by stream filtering is optimal with respect to the input stream: all rays from the stream that would perform the same sequence of operations will always perform those operations together. This observation holds regardless of the order in which rays occur in the input stream or the sequence of operations that these rays undergo to reach the common operations. Thus, given the same input rays, no existing algorithm will be able to combine more operations of the same kind. Moreover, stream filtering requires neither potentially costly presorting operations nor heuristics to estimate coherence.

9.2.3 Application to Ray Tracing

9.2.3.1 Traversal

For traversal, the input stream is recursively traced through a hierarchical acceleration structure such as a bounding volume hierarchy (BVH). In each traversal step, the stream is tested against the bounding box of the current node, and a stream filter partitions the input stream so that only those rays intersecting the node are included in subsequent traversal

operations. If the output stream is empty, the next node is popped from a traversal stack and the process continues with that node. However, if the output stream contains active rays, the output stream is either intersected with the geometry in a leaf node, or the stream is recursively traversed through child nodes in a front-to-back order. As shown by the pseudocode in Figure 9.1, BVH traversal can be written very compactly with stream filtering.

9.2.3.2 Intersection

In their simplest form, stream filters for primitive intersection process an input stream by performing ray/primitive intersection tests in N-wide SIMD fashion. This process generates an N-wide Boolean mask indicating which rays intersect the primitive, and the mask is then used to store intersection information in the ray buffer with conditional scatter operations.

However, rather than perform primitive intersection operations with groups of N elements, the intersection test could instead be decomposed into a sequence of stream filters, or *filter stack*, for the relevant substages. This approach will potentially yield higher efficiency than simply performing the complete intersection test in SIMD fashion. Using a filter stack, each test is applied in succession with only those rays that have passed previous filters, thereby increasing SIMD utilization for a particular input stream. However, ray streams processed during intersection are typically too short to warrant additional filtering operations, particularly for highly complex models, so our implementation does not employ filter stacks for primitive intersection and instead relies on the simpler approach described

```
traverse(node, in_stream)
{
  BoxTest node_test(node);
  out_stream = filter<node_test>(in_stream)
  if (empty(out_stream))
    return

  if (is_leaf(node))
    intersect(primitives, out_stream)
  else
    traverse(front_child(node), out_stream)
    traverse(back_child(node),  out_stream)
}
```

Figure 9.1. Traversal in a BVH with stream filtering. In each traversal step, inactive rays are filtered from the stream before it is forwarded to subsequent operations with the relevant BVH nodes.

above.

9.2.3.3 Shading

Similarly, material shaders could process an input stream by simply performing operations in N-wide SIMD fashion. However, to maintain higher SIMD efficiency, filter stacks are used to extract rays requiring the same operations from the input stream. For example, the complete filter stack for an ideal Lambertian material model consists of six substages. In the stack, input rays are handled by stream filters that extract and process shadow rays, rays that do not intersect geometry, and rays intersecting a light source, each as separate substreams. Any remaining rays are then processed by the material shader, which adds secondary rays as necessary. Additional filtering operations can be applied within each shader to group similar operations; for example, the Lambertian shader probabilistically samples either direct or indirect illumination, and the corresponding ray data are extracted using additional stream filters so that the required operations are performed together.

9.2.4 Programming Model

Using the framework provided by the simulator (described below and shown in Figure 9.2), programmable stream filters are implemented as C++ class templates, and export an interface to generate an output stream corresponding to the active partition. Ray streams are processed in parallel by N-wide SIMD units and are then partitioned into active and inactive subsets before subsequent processing. The *partition* operation employs a comparison sort to move active elements to the start of the stream, and the resulting output stream includes only those elements that pass the corresponding test.

Filter tests are implemented as C++ functors and serve as the template parameter to stream filter objects. Typically, these tests utilize gather operations to process rays in N-wide SIMD units and return a mask indicating the result for each element. Filter tests that modify rendering state use conditional scatter operations to update ray data.

9.3 StreamRay Architecture Description

Figure 9.3 shows the StreamRay architecture. The ray engine consists of efficient address generation mechanisms to support stream assembly. This engine is programmed using C++ templates and supervises data movement for stream assembly. The filter engine consists of N program controlled kernel accelerators that implement each of the ray tracing kernels (traversal, intersection, and shading) in a N-wide SIMD environment.

Stream filter `apply` function template

```
RayStream StreamFilter::apply(RayStream& stream) const
{
  bool  out_mask[MAX_STREAM_LEN] = {};
  bool* mask_end = out_mask;

  int* in_begin  = stream.begin;
  int* in_end    = stream.end;
  int* out_begin = in_begin;
  int* out_end   = out_begin;

  for (int* block = in_begin; block < in_end; block += SIMD_WIDTH)
  {
    const int     nremain = (in_end - block);
    const simd_it duplicate(block[nremain-1]);
   const simd_bt active  = (get_ids<SIMD_WIDTH>() < nremain);
    const simd_it ids     = ifthen(active, (simd_it)block, duplicate);
    const simd_bt mask    = active && test(stream, ids);

    if (anytrue(mask)) store(mask_end, mask);
    mask_end += SIMD_WIDTH;
  }

  out_end += partition(stream.begin, stream.size(), out_mask);
  return RayStream(stream, out_begin, out_end);
}
```

Ray/box filter test

```
simd_bt BoxTest::operator()(const RayStream& stream,
                            const simd_it& ids) const
{
  const simd_ft org_x = gather(stream.in->org_x, ids);
  const simd_ft org_y = gather(stream.in->org_y, ids);
 const simd_ft org_z = gather(stream.in->org_z, ids);
  const simd_ft inv_x = gather(stream.in->inv_x, ids);
  const simd_ft inv_y = gather(stream.in->inv_y, ids);
  const simd_ft inv_z = gather(stream.in->inv_z, ids);
  const simd_ft min_t = zero;
  const simd_ft max_t = gather(stream.in->t_min, ids);
  return box.intersect(org_x, org_y, org_z, inv_x, inv_y, inv_z,
                       min_t, max_t);
}
```

Figure 9.2. Programming model for Stream Filtering. Programmable stream filters export an interface to generate output streams. Filter tests perform the necessary operations and return a mask indicating whether or not individual rays pass the test.

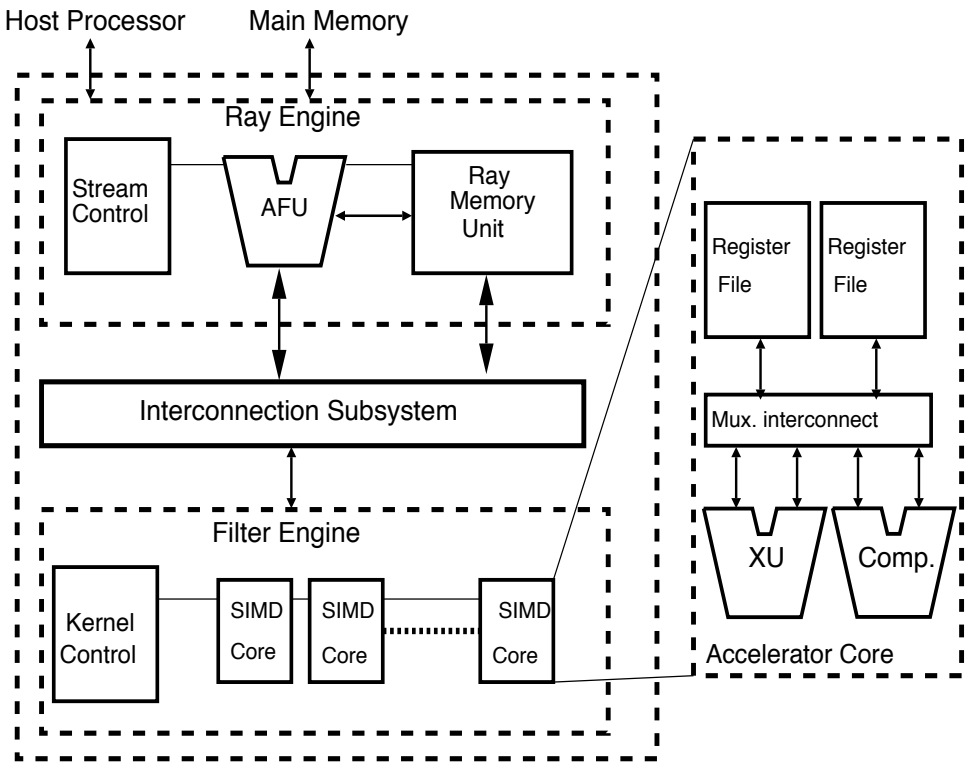

Figure 9.3. StreamRay: High-level view of the N-wide architecture

Kernels are compiled from C++ to object code for the accelerator using the CoGenE compilation framework [77, 78]. The compiler does fine-grain scheduling of the data movement in the programmable interconnect, which is performance critical. The stream control block supervises the two engines and is responsible for synchronization.

9.3.1 The Ray Engine

The ray engine (Figure 9.4) consists of two subsystems: the address fetch unit (AFU) and the ray memory unit. To form a sequential stream of data, N nonsequential memory addresses need to be computed. The address fetch unit consists of N address generator units (AGUs) [77] that provide support for scatter/gather, strided, or sequential addressing. Each AGU is supported by an integer affine function unit and a small register file. The ray memory unit is a distributed system that consists of two ray buffers and a dual-ported scratch pad memory for storing texture data. The ray buffers facilitate data movement between the main memory and the filter engine, so StreamRay employs two such buffers for decoupling: one for current active ray stream, and one that will be used in the next epoch as the active ray stream. This allows the next ray stream to be gathered in parallel with filter processing on the current ray stream. For a stream of size 64×64 rays, it was

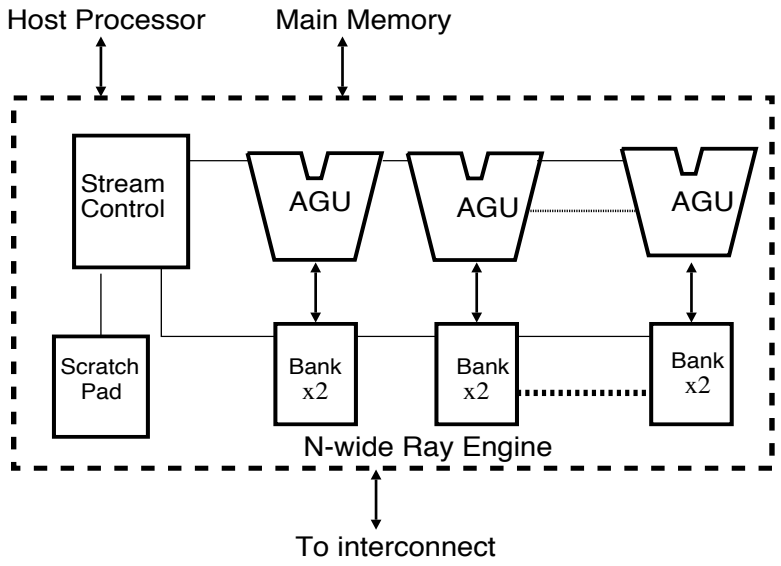

Figure 9.4. Ray architecture: The ray engine provides address computation capabilities and delivers data efficiently to the filter cores

empirically found that a 512 KB buffer provides the best balance between performance and area. Increasing the size beyond 512 KB provides a marginal improvement in the hit rate, and thus rendering performance, but at the cost of increased complexity. To provide support for efficient N-wide SIMD processing, each of the ray buffers are banked into N single-ported ways and each of the AGUs fetch data to one bank. Banking is an efficient alternative to multiported buffers, which are expensive in terms of area and power. Provided that requests do not collide frequently, this design efficiently provides data to the filter engine. Performance is improved as a result of minimized communication and efficient isolation between stream formation and kernel computations.

As an alternative, integer units that perform address computations can be placed in the execution subsystem similar to a traditional processor such as the x86. This approach has the disadvantage that both data and addresses must share the system interconnect and register file. Contention for resources not only degrades performance but also increases the pressure on the compiler to perform efficient data scheduling. As will be shown by our results (Section 9.4), isolating address and data computations improves performance by at least 48% when compared to machines such as the G80 [68], or the R770 [4] that place address processing in the execution subsystem.

9.3.2 The Filter Engine

The filter engine implements the filter operations that partition the stream of rays into active and inactive subsets to exploit the coherence via wide SIMD processing. A set of N accelerators implement the various kernels in ray tracing, including traversal, intersection, and shading. The architecture of the accelerator is shown in Figure 9.3. Each accelerator contains two sets of register files and a set of execution units and comparators. The execution units implement the ray tracing kernels, while the comparators partition the input set into active and inactive subsets. The execution units provide direct support for $+/-$, $*$, $\sqrt{x}$, $\frac{1}{x}$ and bit-masking operations and process operands in SIMD or scalar fashion. As shown in Figure 9.5, the execution units and comparators are backed by pipelined registers and a multiplexer-based interconnect, and can be configured by the program on a cycle-by-cycle basis. The result is a programmable accelerator whose energy-delay characteristics approach that of an ASIC [77, 78].

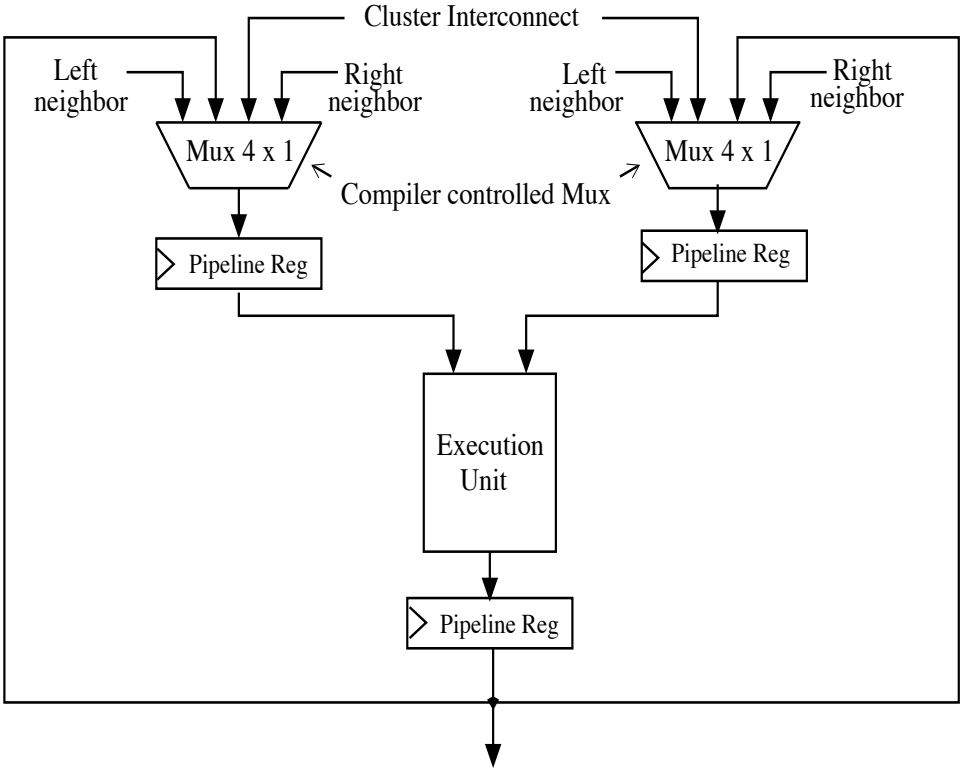

Figure 9.5. Execution unit architecture: Execution units/comparators communicate with the register files through the program-controlled interconnect

9.3.3 Interconnect Subsystem

The interconnect subsystem coordinates data movement across the two engines and is critical to the performance of the system. The StreamRay architecture consists of a simple multiplexer-based interconnect in which each of the accelerators or banks can transmit data in a single cycle to either of its neighbors (left or right). When compared to a fully-connected $N \times N$ network, performance degrades by only 4%, but area is reduced by at least 3× [7, 77]. On the other hand, the simplest possible interconnect uses 1 : 1 mapping between the ray buffer and the kernel accelerators. While this interconnect delivers good utilization for traversal, performance degrades for the intersection and shading operations. As shown in Table 9.1, this network degrades SIMD utilization for the N filter cores by 30% for the different scenes, on average. This result is largely due to the fact that the intersection computation necessitates data transfers from adjacent banks and explicit bank-to-bank copies have to be performed before the data can be used. We thus employ a simple $3 \times N$ network for all our evaluations.

The stream control block issues the load/store memory operations and supervises synchronization for the architecture. While the next-generation ray buffer is filled with data, the current generation ray buffer is used by the filter engine. The kernel control block synchronizes the accelerators across the kernel boundaries. For ray tracing operations that are split into subkernels (for example, shading operations), synchronization occurs across the subkernel boundary. For each kernel, the kernel control block waits until all the filter cores complete the current phase before beginning the next phase. The macroscopic view of the StreamRay architecture is that it is a 2-stage pipeline consisting of a ray engine and a filter engine. However, synchronization of the two stages is somewhat decoupled since the ray engine fills the next ray buffer while the filter engine is operating on the current ray buffer.

Table 9.1. Comparing interconnect choices: Relative performance and area comparisons showcase the benefits of employing a nearest neighbor interconnection strategy

Interconnect type	Performance	Area
Fully connected	1.04	3.0
Neighbors	1.00	1.0
Simple	0.70	0.5-0.6

9.4 Results

A cycle-accurate simulator similar to the SimpleScalar tool set [16] is used to evaluate the architecture. Stalls resulting from data alignment operations are modeled accurately, as is interconnect, function-unit, and memory contention. The multiplexers are carefully sized to allow for single-cycle operation at a frequency of 1 GHz. The address fetch and execution units are synthesized to operate at 1 GHz with 90 nm technology [63]. However, the fully synthesized flexible interconnect will not run at this frequency. Significant manual design might will achieve the 1 GHz target but we have yet to prove this conjecture. Hence, the subsequent design analysis is based on an achievable 500 MHz clock in a 90 nm process. A summary of the architectural parameters is shown in Table 9.2.

9.4.1 Methodology

Images are generated using a Monte Carlo path tracer [48] compiled for the simulated N-wide SIMD architecture. Currently, the renderer supports three different material models: a coupled model for glossy reflections, dielectric materials such as glass and ceramic, and ideal Lambertian surfaces. The renderer also uses a thin-lens camera model to simulate depth-of-field effects. Ray streams are traced in a breadth-first manner: primary rays are traced to completion, populating an output buffer with secondary rays as necessary. Pointers to the input and output buffers are swapped, and each subsequent generation of rays is traced in a similar manner. This process continues until the input stream contains no elements. This study renders three scenes of varying geometric complexity, visual effects, and shader types, and the details of each are summarized in Table 9.3.

Table 9.2. Architecture and rendering parameters

Parameter	**Value**
Process	90nm (500 MHz-1 GHz)
SIMD width	8, 12, 16
Buffer size	256, 512, 1024 KB each
Number of banks	8, 16, 32, 64
Buffer access	2 cycles
Multiplexer width	4 (max)
Interconnect levels	2, 4
Ray stream size	4 KB (32×32) , 16 KB (64×64)
Image resolution	1024×1024 pixels
Samples per pixel	64

Table 9.3. Characteristics of the test scenes: Scenes of varying geometric complexity are used to evaluate the potential role of stream filtering in interactive ray tracing. These scenes employ three different material shaders to capture a variety of important visual effects.

			material shaders			per-frame stats		
scene	**# prims**	**# lights**	**coupled**	**dielectric**	**lambert**	**# rays**	**# trav ops**	**# isec ops**
rtrt	83845	2	•	•	•	1.18×10^8	9.77×10^6	1.26×10^6
conf	282644	72	•		•	1.62×10^8	3.25×10^8	6.51×10^7
kala	2124001	2			•	1.57×10^8	7.63×10^8	9.01×10^7

9.4.2 Evaluation

The performance of the StreamRay approach is evaluated in terms of SIMD utilization and frame rate. SIMD utilization indicates how efficiently the N-wide SIMD units are used, and frame rate is a measure of rendering performance in frames/second. Although SIMD utilization is reported for primary and secondary rays for all three stages of rendering (traversal, intersection, and shading), algorithms that normally work well with primary rays have been shown to perform poorly with secondary rays [84]. Hence, the particular emphasis is on secondary ray performance on the StreamRay architecture.

9.4.2.1 SIMD Utilization

High utilization rates are observed for the three scenes for primary rays. For traversal, utilization is as high as 95% for a SIMD width of 8 and marginally reduces to around 91% for a SIMD width of 16. For intersection, utilization rates are approximately 90%. As the initial stream size increases, utilization increases significantly because inactive rays are automatically removed from the output stream. There are two possible sources of bottlenecks: first, input streams in general are not multiples of the SIMD width and so the last SIMD operation may be partially filled; and second, insufficient coherence can deliver substreams that are shorter than the SIMD width. Utilization drops below 100% in both these cases.

SIMD utilization for traversal, intersection, and shading (T / I / S) in the path tracer for secondary rays is shown in Table 9.4. We employ 64 samples per pixel to approximate rays that might be generated in practice. Utilization remains reasonably high under a variety of SIMD widths, with larger initial ray streams again leading to higher utilization in all stages for all scenes. Compared to an oracle system with no stalls, utilization degrades by 5-10% for traversal, 10-21% for intersection, and 2-5% for shading. Stalls arise because of overheads introduced by address fetch and alignment in the ray engine, and by data partitioning in the filter engine.

The data show that complex scenes with many small triangles lead to lower utilization

Table 9.4. SIMD utilization (%) (T / I / S) for secondary rays: Stream filtering exploits any coherence available in a particular stream

size	rtrt	conf	kala
	SIMD width $N = 8$		
32×32	70 / 47 / 86	57 / 23 / 90	56 / 26 / 96
64×64	77 / 55 / 93	70 / 31 / 95	67 / 31 / 95
	SIMD width $N = 12$		
32×32	60 / 38 / 82	45 / 17 / 89	45 / 19 / 92
64×64	70 / 46 / 92	61 / 24 / 96	56 / 25 / 97
	SIMD width $N = 16$		
32×32	52 / 34 / 81	38 / 13 / 87	38 / 14 / 91
64×64	65 / 42 / 89	54 / 18 / 94	49 / 20 / 96

during traversal and intersection. Intersection suffers the greatest reduction in stream length. In contrast, shaders typically possess the longest streams and splitting the shading operation into many subkernels results in high utilization. Overall, stream filtering successfully extracts any coherence exhibited by rays in a particular stream.

9.4.2.2 Rendering Performance

In general, StreamRay delivers interactive frame rates (above 10 fps) for the test scenes. As shown in Table 9.5, performance increases with the SIMD width, due to the reduced number of alignment and partitioning operations. On the other hand, wider SIMD units require more time for address computation and have higher address fetch overhead. The inherent trade-off between SIMD width and fetch overhead is the fundamental performance constraint of the StreamRay approach.

For *rtrt*, a SIMD width of eight balances the overheads sufficiently, and performance

Table 9.5. Rendering performance: StreamRay delivers interactive frame rates for all scenes

size	rtrt	conf	kala
	SIMD width $N = 8$		
32×32	16.60 fps	8.15 fps	6.73 fps
64×64	18.78 fps	12.78 fps	8.34 fps
	SIMD width $N = 12$		
32×32	21.82 fps	12.56 fps	11.78 fps
64×64	24.52 fps	18.32 fps	13.45 fps
	SIMD width $N = 16$		
32×32	22.36 fps	14.35 fps	13.34 fps
64×64	26.35 fps	20.32 fps	15.65 fps

exceeds the 10 fps threshold. However, in moving from 8-wide to 12-wide SIMD units, significant improvements are observed for all three scenes due to reduced SIMD alignment and stream partitioning overhead. Beyond a width of 12, the overhead of address computation begins to dominate, and improvements diminish accordingly. Thus, a 12-wide SIMD machine is the optimum choice in this case-study.

These results demonstrate that StreamRay achieves interactive frame rates for complex scenes using path tracing and a variety of visual effects. As processors continue to rely on increasing levels of fine-grained parallelism, we believe that hardware support for wider-than-four SIMD processing and nonsequential memory access will become commonplace. With these architectures, stream filtering architectures become a viable alternative for interactive ray tracing.

9.4.3 StreamRay Efficiency

Evaluating the different subsystems within the StreamRay architecture is not straightforward. However, in this section, each subsystem under examination is compared against an oracle best-case implementation or an existing feasible implementation.

9.4.3.1 Address Processing vs. Data Processing

Table 9.6 shows the distribution of major operations types for a stream of size 64×64 elements and a SIMD width of 16. In these data, the integer operations required by address fetch are subsumed by load and store operations. While varying SIMD widths change the absolute number of operations for a given frame, the ratios are preserved. It can be observed that data computations account for as much as 31-35% of the total operations. Address computations also account for a similar fraction (28-34%).

These results make a compelling argument for supporting both address and data computations efficiently. StreamRay isolates these computations by efficiently decoupling the memory system from the execution system and placing the integer execution unit in the address generation unit. AGUs need to perform integer computations to support scat-

Table 9.6. Distribution of major operations as % of total: Here, the compute-related operations refer to those involving actual ray data; integer operations are subsumed by the load and store operations.

scene	load	store	comp	scat/gath	part
rtrt	24.1	14.9	35.4	19.6	5.0
conf	23.2	19.7	34.5	18.7	3.8
kala	23.8	20.3	31.9	21.7	2.0

ter/gather operations on-the-fly, and this approach reduces data movement and contention for those shared resources (interconnect, register files, etc.) that would otherwise be used if the integer units were placed in the execution subsystem. The latter approach is common in traditional processors and is also employed in current machines like the G80 and the R770. In addition to causing contention for shared resources, this increases the burden on the compiler to generate efficient code.

To evaluate the performance benefit of moving integer execution units to the ray engine, StreamRay is compared against an architecture in which the integer units are placed in the filter engine and each of the AGUs are paired with the integer units. The overall performance improvement and reduction in power dissipation per filter core for each of the scenes is shown in Table 9.7. As compared to a traditional execution core, StreamRay delivers an average 56% performance speedup. The speedup is higher for complex scenes like *conf* and *kala* [36]. This effect can be attributed to the increased dependence on address computations for intersection and for supporting high-quality visual effects during the shading process. In addition, reduced data movement and contention provide power savings of 11.63% for each accelerator core. Thus, placing integer units intelligently provides performance and power benefits while also reducing programming complexity.

9.4.3.2 Partitioning Efficiency

The design of the filter engine is critical to sustaining the parallelism generated by the ray engine. Each of the accelerators implement the filter kernel and the partitioning operation. Though the partitioning operation accounts for only 2-5% of the total operations, it is in the critical path for each operation: for ray tracing operations such as traversal, each step requires that an input stream be partitioned into an active and inactive subset. During partitioning, the Boolean mask is checked and rays pointers are updated to indicate if they

Table 9.7. Isolating address and data computations: StreamRay delivers higher performance at reduced power dissipation over a traditional execution subsystem by placing integer execution units in AGUs

Parameter	***rtrt***	***conf***	***kala***
	SIMD width $N = 12$		
Performance speedup	1.50	1.67	1.53
Power savings/filter core (%)	12.2	13.3	9.4
	SIMD width $N = 16$		
Performance speedup	1.48	1.63	1.49
Power savings/accelerator core (%)	11.2	13.1	8.4

are active or not. The ray data are then moved to one of the register files and the active and inactive set pointers are updated. In the case of an empty active set, the next node is fetched from the traversal stack for further operations. The performance of the filter core in efficiently implementing the partition operation can be evaluated by comparing against an oracle scheme assuming no overhead for partitioning. We observe that the performance of the filter engine suffers by 2-4% for traversal, 7-12% for intersection, and 1-2% for shading. The partitioning overhead is negligible for both traversal and shading. For intersection, if there is little coherence within a stream, it causes repeated updates of the active and inactive set pointers and subsequently, leads to data movement overhead.

9.4.3.3 Frequency Scalability of Interconnect

As noted, the 500 MHz clock frequency was primarily constrained by the interconnection subsystem. There are two key issues that influence the design. Increasing the multiplexer width will increase the number of comparison operations which will increase interconnect delay. Increasing the frequency of the interconnect may require the insertion of pipeline registers, which will increase the network latency in terms of clock cycles.

The frequency sensitivity of StreamRay can be evaluated by increasing the delay through the interconnect for a higher operating frequency. For a 50% increase in frequency (750 MHz), delay through the interconnect subsystem is doubled by introducing pipelined registers. The resulting frame rates for the test scenes are shown in Table 9.8, and rendering performance scales up by around 27% on an average. It is interesting to note that both *rtrt* [36] and *kala* scale marginally better than the *conf* scene. Doubling the delay of the interconnect increases the scheduling conflicts on both the stream control and the kernel control block. Overheads for partitioning increases to 12-15% for the intersection computation and this contributes to some of the scaling degradation.

9.4.3.4 Supporting Alternative Ray Tracing Algorithms

The stream filtering approach discussed in this study generalizes several other techniques in the ray tracing literature. In particular, by employing appropriate values for stream length and SIMD width, stream filtering can be used to implement standard recursive ray tracing [101] or packet-based ray tracing [99]. We also believe that similar opportunities exist for other ray tracing algorithms.

Table 9.8. Frequency scalability: Rendering performance scales well when the interconnect delay is doubled for a 50% increase in operating frequency

size	**rtrt**	**conf**	**kala**
	SIMD width $N = 8$		
32×32	21.08 fps	11.04 fps	10.05 fps
64×64	24.78 fps	15.97 fps	12.34 fps
	SIMD width $N = 12$		
32×32	28.02 fps	15.56 fps	14.78 fps
64×64	31.52 fps	22.32 fps	16.85 fps
	SIMD width $N = 16$		
32×32	29.56 fps	17.85 fps	17.34 fps
64×64	32.35 fps	24.92 fps	20.65 fps

CHAPTER 10

CONCLUSIONS AND FUTURE WORK

With the advent of information explosion and fusion, the definition of what constitutes an embedded computing system is expanding and an embedded device is expected to provide a plethora of services to the end user. As the user demands more applications on a single device, the amount of resources, expertise, and time for designing such a device will increase significantly. Given the strict constraints imposed by the business of embedded markets, this problem has created many challenges for application and compiler experts, VLSI engineers, and system designers. This dissertation presents CoGenE, a framework that automates the design of programmable energy-performance optimal DSAs for embedded systems. CoGenE can be used by application experts who have little/no knowledge in the areas of compilers, architecture, or circuit design. Given an application domain, the application expert can employ CoGenE to explore a variety of design choices based on performance, power, energy, area, and programmability and pick the architecture of his or her choice. CoGenE also delivers a compiler that generates object code for the selected architectural candidate. With traditional techniques, designing a compiler and an architecture for an application domain involves man-months of time and valuable resources. As demonstrated in Chapter 8, the application expert can generate the compiler and the energy-performance optimal DSA in hours or days. CoGenE is a new design methodology that represents a significant improvement in performance, energy dissipation, design time, and resources.

In addition to designing DSAs for embedded systems, CoGenE can be employed to design constituent parts of highly parallel multiprocessor systems. The versatility of this approach was demonstrated in Chapter 9, where CoGenE was employed to automatically synthesize the compiler and the SIMD core for a N-wide multicore SIMD architecture. This dissertation also presents StreamRay, an novel architecture for computer graphics. The key idea is that executing address and data computations separately in space simultaneously reduces data communication and contention for resources, thereby delivering a performance that is significantly better than current ray trace processors. CoGenE was also employed

to synthesize the interconnection subsystem within StreamRay. Overall, this demonstrates the robustness and versatility of CoGenE in creating high performance DSAs in various application domains.

10.1 Contributions

The contributions of this dissertation in the area of embedded systems are:

10.1.1 CoGenE

A novel design framework that automates the design of DSAs by automatically generating a compiler, an energy-performance optimal DSA, and a host of DSAs that satisfy various user defined constraints. Design time is on the order of hours or days and represents a significant improvement over man-months of manual design time. Further, an application expert can employ CoGenE to survey the entire design space. CoGenE is thus a modular framework for embedded DSA design.

10.1.2 "Interconnection-aware" Compilation

CoGenE explores the design space of "ASIC-like" DSAs due to their superior performance and energy characteristics. Chapter 5 demonstrated that scheduling data on the interconnect is key to the performance of such DSAs. The CoGenE compiler employs ILP-based interconnection scheduling techniques to generate execution binaries that deliver high performance at very low energy dissipation. Code generation time is on the order of tens of minutes or hours and removes the need to perform error-prone manual code generation, as is the case in many embedded systems today.

10.1.3 Design Space Exploration

For an application expert to design a DSA, the framework should automatically search the architectural design space to select the best candidate. The CoGenE design explorer employs SCA, an iterative search technique to survey the entire design space efficiently. It provides the application expert with a feasible set of design choices. This process incurs hours and removes the need for architecture expertise, thereby reducing design time and valuable resources.

10.1.4 Face Recognition Characterization

To our knowledge, this is the first study that characterizes the computational requirements of a variety of face recognition algorithms. Two recognition algorithms, the PCA/LDA

and the EBGM algorithm, were analyzed and it was found that the PCA/LDA algorithm was more amenable to deployment in embedded devices.

10.1.5 The CoGenE Power Simulator

To accurately estimate the power dissipation in embedded systems, the simulator employs empirical models for complex circuits like AGUs, HLUs, interconnects, and ALUs. Power dissipation is a first-order constraint and is critical to arrive at energy-performance optimal designs.

10.1.6 CoGenE for Ray Tracing

Stream filtering is a new approach to ray tracing which creates arbitrarily sized groups of coherent rays to efficiently utilize wider-than-four SIMD units. StreamRay efficiently isolates data and address processing to deliver the parallelism required for interactive ray tracing. The major advantages of this approach are:

- **Parallel processing** The algorithm achieves high SIMD utilization by exploiting the parallelism inherent to any collection of rays. StreamRay provides the capability for interactive rendering by efficiently implementing the algorithm.
- **Implicit reordering** The algorithm extracts active rays with respect to scene geometry, acceleration structure, material shaders, and so forth, and does not depend on presorting operations or ray coherence heuristics.
- **Generality and scalability** The algorithm is generally applicable to all hierarchical acceleration structures and any type of primitive, and thus supports a wide range of ray tracing applications.

The StreamRay architecture provides hardware support for this approach, and results demonstrate that this technique delivers high performance and also opens up a new design space for ray tracing accelerators.

10.2 Future Work

10.2.1 Code Splitting

As described in the grand goal, the immediate future work is to automate the process of code splitting. Every application has to be split into sequential code and parallel streaming code before we can map the different pieces of code to different processors. Automatic code splitting is an application-dependent task and it is likely that an interactive tool that aids

the application expert may be of great use. This tool will reduce the cost and level of expertise required to split code into sequential and streaming code.

A simple way to identify the compute-intensive kernels is to profile the code for processing time and energy. This technique was employed in Chapter 3 to identify the various phases of face recognition. Once the parallel code is identified, the original application can be partitioned with little effort. Manually annotating the application with interface code that facilitates communication between the GPP and the DSA will help in code splitting.

10.2.2 Integrated "Interconnect-Register" Scheduling

In our current compiler flow, Register and interconnect scheduling can be done in either order and the second process is limited by decisions made in the first. An integrated approach will likely add a modeling constraint to the ILP formulation and may lead to increased compilation time. The register file can be treated as a partitioned system that is paired with interconnects that are closely located in space. This will reduce the complexity of the modeling constraint and produce better code schedules.

10.2.3 Automatic Code Verification

Our current infrastructure checks the correctness of generated code by comparing the results of the cycle accurate simulator against a functional simulator. In recent years, there has been a lot of interest in verifying the correctness of compiler generated binary for reliability critical applications. As devices expand to perform bio-medical applications like heart rate monitoring, automatic code verification will become mandatory and is an important area for future research.

10.2.4 Emerging Application Domains

In the near future, the framework will be employed to design DSAs for two application domains that are becoming increasingly important:

- *Automotive Engineering*: Vehicles that are manufactured today contain many microcontrollers and processors to perform common operations like ABS, traction control, detecting sleepiness, etc. These operations are well-suited to domain-specific acceleration and fit a stream processing model discussed in this study. Given the high volume and the huge application space, DSAs for this domain need to be researched.

- *Finance Modeling*: Another application domain which will significantly benefit from acceleration is finance modeling. Investigating macro and micro-economic trends

in various business is compute intensive and time consuming. This is a significant wastage in power requirements and manpower. Given the importance of accelerating economic trends, this domain remains an active research area for performance improvements.

10.2.5 Ray Tracing

CoGenE's evaluation on ray tracing has opened many new avenues for investigation. Some of the major challenges are:

- **Multiprocessing** Design choices include heterogeneous and homogeneous systems. In the former, each core might be responsible for one particular stage, communicating with other cores via a dual-buffered output memory, which permits data to be simultaneously read by the next core in the pipeline. In a homogeneous multicore system, a single core can be replicated to provide additional more coarse-grained levels of parallelism. In addition to low design complexity and core scalability, the latter requires only minimal changes in the execution subsystem.

- **Memory hierarchy** The operations in ray tracing exhibit different access patterns. The intersection computation employs the neighbor banks to extract performance out of the system. Designing a custom hierarchy for each operation can lead to a highly optimized heterogeneous system.

- **Operations besides traversal, intersection, and shading** Stream filtering can also be applied to other operations not strictly related to ray tracing, provided these operations employ a hierarchical data structure and can be written in SIMD fashion. Thus, stream filtering can potentially be used with a wide range of other rendering algorithms as well.

- **Nontraditional hardware architectures** We believe that the stream filtering approach may also be well-suited to architectures such as Imagine [50], Merrimac [26], or perhaps Intel's upcoming Larrabee processor [90]. Results obtained with StreamRay also make a compelling argument to further investigate designs for special-purpose, ray-based graphics hardware.

- **Hardware support for special operations** The accelerator cores that implement the filtering and partitioning operations are general enough to support a variety of other operations. As processors begin to rely on fine-grained parallelism, custom hardware support for such operations may be critical for energy-delay efficiency.

- **Specialized ALUs** The execution units employ generalized floating point units. To further improve energy efficiency, specialized execution units can be explored for each of these domains. For example, applications that only require multiplication in powers of 2 do not require a multiplier circuit. A left-shift implementation would be more energy efficient.

- **Schemes for intersection** Instead of processing streams of rays that require intersection with the same primitive, the algorithm could be modified to combine substreams for different primitives. StreamRay provides the support to enable each stream element to process a different primitive, but merging the results would require additional hardware support.

The architecture and the supporting framework presented in this work provides a compelling design for future ray-based graphics hardware, and we plan to explore both heterogeneous and homogeneous multicore designs to support renderers based on stream filtering. We also plan to explore real-time implementations of the algorithm with current processors in an attempt to eliminate the need for hardware simulation. With increasing support for SIMD parallelism expected in new generations of commodity architectures, we hope to achieve real-time performance with stream filtering for a wide variety of rendering algorithms.

REFERENCES

[1] Abraham, S. G., and Rau, B. R. Efficient design space exploration in pico. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems* (New York, NY, USA, 2000), ACM, pp. 71–79.

[2] Akturan, C., and Jacome, M. F. FDRA: A software-pipelining algorithm for embedded VLIW processors. In *Proceedings of the International Symposium on System Sythesis* (2000), pp. 34–40.

[3] Akturan, C., and Jacome, M. F. Caliber: A software pipelining algorithm for clustered embedded VLIW processors. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)* (2001), pp. 112–118.

[4] ATI. ATI products from AMD. http://ati.amd.com/products/index.html.

[5] Balasubramonian, R., Muralimanohar, N., Ramani, K., and Venkatachalapathy, V. Microarchitectural Wire Management for Performance and Power in Partitioned Architectures. In *Proceedings of HPCA-11* (February 2005).

[6] Banerjee, K., and Mehrotra, A. A Power-optimal Repeater Insertion Methodology for Global Interconnects in Nanometer Designs. *IEEE Transactions on Electron Devices 49*, 11 (November 2002), 2001–2007.

[7] Benthin, C. *Realtime Ray Tracing on Current CPU Architectures.* PhD thesis, Saarland University, 2006.

[8] Bertran, A., Yu, H., and Sacchetto, P. Face detection project report. http://-ise.stanford.edu/2002projects/ee368/Project/reports/ee368group17.pdf, 2002.

[9] Bluespec. Bluespec modeling. http://www.bluespec.com.

[10] Bolme, D. S. *Elastic Bunch Graph Matching.* PhD thesis, Colorado State University, June 2003.

[11] Boulos, S., Edwards, D., Lacewell, J. D., Kniss, J., Kautz, J., Wald, I., and Shirley, P. Packet-based Whitted and distribution ray tracing. In *Graphics Interface 2007* (May 2007), pp. 177–184.

[12] Brash, D. The ARM Archtecture Version 6 (ARMv6). ARM Holdings plc Whitepaper, January 2002.

[13] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture ISCA-27* (June 2000).

[14] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA* (2000), pp. 83–94.

[15] Burger, D., and Austin, T. The Simplescalar Toolset, Version 2.0. Tech. Rep. TR-97-1342, University of Wisconsin-Madison, June 1997.

[16] Burger, D., and Austin, T. M. The simplescalar toolset, version 2.0. Tech. Rep. TR-97-1342, University of Wisconsin-Madison, June 1997.

[17] Burns, G., Jacobs, M., Lindwer, W., and Vandewiele, B. Silicon hive's scalable and modular architectural template for high performance multi-core systems. In *GSPx* (Oct. 2005).

[18] Catmull, E. E. *A submdivision algorithm for computer display of curved surfaces.* PhD thesis, University of Utah, 1974.

[19] Chellappa, R., Wilson, C., and Sirohey, S. Human and machine recognition of faces: A survey. In *PIEEE* (May 1995), vol. 83, pp. 705–740.

[20] Clark, N., Tang, W., and Mahlke, S. Automatically generating custom instruction set extensions. In *Proceedings of the Workshop on Application-Specific Processors held in conjunction with the Annual International Symposium on Microarchitecture* (2002).

[21] Clark, N., Zhong, H., and Mahlke, S. Processor acceleration through automated instruction set customisation. In *Proceedings of MICRO* (2003).

[22] Cleary, J., Wyvill, B., Birtwistle, G., and Vatti, R. A parallel ray tracing computer. In *Proceedings of the Association of Simulat Users Conference* (1983), pp. 77–80.

[23] Colorado State University. Evaluation of face recognition algorithms. http://-www.cs.colostate.edu/evalfacerec/, 2003.

[24] Crovella, M. E., and LeBlanc, T. J. Parallel Performance Prediction Using Lost Cycles Analysis. In *Supercomputing '94* (1994), IEEE Computer Society.

[25] D. Bolme, R. Beveridge, M. T., and Draper, B. The CSU face identification evaluation system: Its purpose, features and structure. In *International Conference on Vision Systems* (April 2003), pp. 304–311.

[26] Dally, W., and Hanrahan, P. Merrimac: Supercomputing with Streams. In *Supercomputing* (2003).

[27] Dhodapkar, A., Lim, C. H., and Cai, G. TEM2p2EST: A Thermal Enabled Multi-Model Power/Performance Estimator. In *Proc. Workshop on Power-Aware Computer Systems (PACS'00), Cambridge, MA* (Nov. 2000).

[28] Dubey, P. A platform 2015 workload model: Recognition, mining, and synthesis moves computers to the era of tera. *Intel Corporation White Paper* (Feb. 2005).

[29] Eckstein, E., and Krall, A. Minimizing cost of local variables access for DSP-processors. In *LCTES'99 Workshop on Languages, Compilers and Tools for Embedded Systems* (Atlanta, 1999), Y. A. Liu and R. Wilhelm, Eds., vol. 34(7), pp. 20–27.

[30] EisenBrand, F. *Gomory-Chvatal Cutting Planes and the Elementary Closure of Polyhedra*. PhD thesis, Saarland University, 2000.

[31] Faraboschi, P., Brown, G., Fisher, J. A., Desoli, G., and Homewood, F. Lx: a technology platform for customizable VLIW embedded processing. In *The 27th Annual International Symposium on Computer architecture 2000* (New York, NY, USA, 2000), ACM Press, pp. 203–213.

[32] Fields, B. A., Bodík, R., Hill, M. D., and Newburn, C. J. Using interaction costs for microarchitectural bottleneck analysis. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), IEEE Computer Society, p. 228.

[33] Gonzalez, R., and Horowitz, M. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits 31*, 9 (Sept. 1996), 1277–1284.

[34] Gonzalez, R. E. Xtensa — A configurable and extensible processor. *IEEE Micro 20*, 2 (/2000), 60–70.

[35] Gottumukkal, R., and Asari, K. System level design of real time face recognition architecture based on composite pca. In *GLSVLSI 2003* (2003).

[36] Gribble, C. P., and Ramani, K. Coherent ray tracing via stream filtering. In *2008 IEEE/Eurographics Symposium on Interactive Ray Tracing* (August 2008). To appear.

[37] Grun, P., Dutt, N., and Nicolau, A. Access Pattern-Based Memory and Connectivity Architecture Exploration . *ACM Transactions on Embedded Computing Systems 2*, 1 (February 2003).

[38] Gschwind, M. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming 35*, 3 (2007), 233–262.

[39] Haralick, R., and Shapiro, L. *Computer and Robot Vision*, vol. 1. Addison-Wesley Publishing Company, 1992, ch. 5.

[40] Hoogerbrugge, J., and Augusteijn, L. Instruction scheduling for TriMedia. *Journal of Instruction-Level Parallelism, 1(1)* (Feb. 1999).

[41] Hoogerbrugge, J., Corporaal, H., and Mulder, H. MOVE: a framework for high-performance processor design. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (1991), ACM Press, pp. 692–701.

[42] Ibrahim, A. *ACT: Adaptive Cellular Telephony Co-Processor*. PhD thesis, University of Utah, December 2005.

[43] Ibrahim, A., Parker, M., and Davis, A. Energy efficient cluster co-processors. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* (May 2004).

[44] Ipek, E., McKee, S. A., Caruana, R., de Supinski, B. R., and Schulz, M. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (2006), pp. 195–206.

[45] Jacome, M., and de Veciana, G. Design challenges for new application specific processors, June 2000.

[46] Joshi, S. M., and Dubey, P. Some Fast Speech Processing Algorithms Using Altivec Technology. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* (1999).

[47] Junqua, J. *Robust Speech Recognition in Embedded Systems and PC Applications*, 1st ed. Springer, May 2000.

[48] Kajiya, J. T. The rendering equation. In *Siggraph 1986* (1986), pp. 143–150.

[49] Karkhanis, T. S., and Smith, J. E. Automated design of application specific superscalar processors: an analytical approach. *SIGARCH Comput. Archit. News 35*, 2 (2007), 402–411.

[50] Khailany, B., Dally, W. J., Kapasi, U. J., Mattson, P., Namkoong, J., Owens, J. D., Towles, B., Change, A., and Rixner, S. Imagine: Media processing with streams. *IEEE Micro 21*, 2 (2001), 35–46.

[51] Lai, C., Lu, S.-L., and Zhao, Q. Performance analysis of speech recognition software. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads* (Feb. 2002).

[52] Lapinskii, V., Jacome, M., and de Veciana, G. Application-specific clustered VLIW datapaths: Early exploration 32 on a parameterized design space. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21(8)* (August 2002), 889–903.

[53] Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., and Amarasinghe, S. P. Space-time scheduling of instruction-level parallelism on a raw machine. In *Architectural Support for Programming Languages and Operating Systems* (1998), pp. 46–57.

[54] Leupers, R. Instruction scheduling for clustered VLIW DSPs. In *IEEE PACT* (2000), pp. 291–300.

[55] Liang, B., and Dubey, P. Recognition, mining, and synthesis. *Intel Technology Journal 9*, 2 (May 2005).

[56] Magen, N., Kolodny, A., Weiser, U., and Shamir, N. Interconnect Power Dissipation in a Microprocessor. In *Proceedings of System Level Interconnect Prediction* (February 2004).

[57] Mahovsky, J., and Wyvill, B. Memory-conserving bounding volume hierarchies with coherent raytracing. *Computer Graphics Forum 25*, 2 (2006), 173–182.

[58] Mansson, E., Munkberg, J., and Akenine-Moller, T. Deep coherent ray tracing. In *2007 IEEE Symposium on Interactive Ray Tracing* (September 2007), pp. 79–85.

[59] Mathew, B., Davis, A., and Evans, R. A characterization of visual feature recognition. In *Proceedings of the IEEE 6th Annual Workshop on Workload Characterization (WWC-6)* (October 2003), pp. 3–11.

[60] Mathew, B., Davis, A., and Fang, Z. A low-power accelerator for the Sphinx 3 speech recognition system. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03)* (October 2003), pp. 210–219.

[61] Mathew, B., Davis, A., and Parker, M. A Low Power Architecture for Embedded Perception. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '04)* (September 2004).

[62] Mathew, B. K. *The Perception Processor*. PhD thesis, University of Utah, August 2004.

[63] Mathew, B. K., Davis, A., and Parker, M. A. A low power architecture for embedded perception. In *CASES '04: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (September 2004), pp. 46–56.

[64] Mcfarland, M., Parker, A., and Camposano, R. Tutorial on high level synthesis. In *Design automation conference* (1988).

[65] Mei, B., Vernalde, S., Verkest, D., Man, H. D., and Lauwereins, R. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *FPL'03* (2003).

[66] Nakamaru, K., and Ohno, Y. Breadth-first ray tracing utilizing uniform spatial subdivision. *IEEE Transactions on Visualization and Computer Graphics 3*, 4 (1997), 316–328.

[67] Nguyen, H., and John, L. K. Exploiting SIMD parallelism in DSP and multimedia algorithms using the altivec technology. In *International Conference on Supercomputing* (1999), pp. 11–20.

[68] NVIDIA. NVIDIA GeForce 8800 GPU Architectural Overview. November 2006.

[69] Parcerisa, J.-M., and Gonzlez, A. Multithreaded Decoupled Access/Execute Processors. Tech. rep., Universitat Politecnica de Catalunya, 1997.

[70] Park, H., Fan, K., Kudlur, M., and Mahlke, S. Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures. In *CASES* (October 2006).

[71] Parker, S., Martin, W., Sloan, P.-P. J., Shirley, P., Smits, B., and Hansen, C. Interactive ray tracing. In *Symposium on Interactive 3D Graphics* (1999), pp. 119–126.

[72] Phillips, P., Moon, H., Rizvi, S., and Rauss, P. The feret evaluation methodology for face-recognition algorithms. In *T-PAMI* (October 2000), vol. 22, pp. 1090–1104.

[73] Pozzi, L., and Ienne, P. Exploiting Pipeline to Relax Register-File Port Constraints of Instruction Set Extensions. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (September 2005), pp. 2–10.

[74] Press, T. A. Techies dissect apple's iphone.

[75] Rabaey, J. M. Wireless beyond the third generation - facing the energy challenge.

[76] Rabaey, J. M., Potkonjak, M., Koushanfar, F., Li, S., and Tuan, T. Challenges and opportunities in broadband and wireless communication designs. In *ICCAD* (November 2000), pp. 76–83.

[77] Ramani, K., and Davis, A. Application driven embedded system design: A face recognition case study. In *CASES '07: Proceedings of the 2007 International conference on compilers, architectures, and synthesis for embedded Systems* (2007), pp. 103–114.

[78] Ramani, K., and Davis, A. Automating the Design of Embedded Domain Specific Accelerators. Tech. rep., University of Utah, 2008.

[79] Ramani, K., Gribble, C. P., and Davis, A. Elided for blind review.

[80] Ramani, K., Gribble, C. P., and Davis, A. Stream filtering with streamray: An architecture for coherent ray tracing. Tech. Rep. GCC-CS-002-2008, Grove City College, 2008.

[81] Ramani, K., Ibrahim, A., and Shimizu, D. PowerRed: A Flexible Modeling Framework for Power Efficiency Exploration in GPUs. In *Proceedings of the Workshop on General Purpose Processing on GPUs, GPGPU'07.*

[82] Ramani, K., Muralimanohar, N., and Balasubramonian, R. Microarchitectural Techniques to Reduce Interconnect Power in Clustered Processors. In *Proceedings of the 5th Workshop on Complexity-Effective Design, held in conjunction with ISCA-31* (June 2004).

[83] Rau, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (1994).

[84] Reshetov, A. Omnidirectional ray tracing traversal algorithm for kd-trees. In *2006 IEEE Symposium on Interactive Ray Tracing* (September 2006), pp. 57–60.

[85] Rixner, S., Dally, W. J., Kapasi, U. J., Khailany, B., Lopez-Lagunas, A., Mattson, P. R., and Owens, J. D. A bandwidth-efficient architecture for media processing. In *International Symposium on Microarchitecture* (1998), pp. 3–13.

[86] Rowley, H. A., Baluja, S., and Kanade, T. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence 20*, 1 (1998), 23–38.

[87] Schapire, R. E. The boosting approach to machine learning: An overview. In *In MSRI Workshop on Nonlinear Estimation and Classification* (2002).

[88] Schlansker, M. S., and Cover, B. R. R. EPIC: Explicitly parallel instruction computing. *Computer 33*, 2 (2000), 37–45.

[89] Schmittler, J., Wald, I., and Slusallek, P. SaarCOR: A hardware architecture for ray tracing. In *Eurographics Workshop on Graphics Hardware* (September 2002), pp. 27–36.

[90] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3 (2008), 1–15.

[91] Semiconductor Industry Association. International Technology Roadmap for Semiconductors 2005. `http://www.itrs.net/Links/2005ITRS/Home2005.htm`.

[92] Silvano, C., Agosta, G., and Palermo, G. Efficient architecture/compiler co-exploration using analytical models. *Design Automation for Embedded Systems 11*, 1 (2007), 1–25.

[93] Smith, M. D., Lam, M., and Horowitz, M. A. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual Symposium on Computer Architecture* (1990), pp. 344–354.

[94] Soriano, M., Martinkauppi, B., Huovinen, S., and Laaksonen, M. Using the skin locus to cope with changing illumination conditions in color-based face tracking. In *Proceedings of the IEEE Nordic Signal Processing Symposium* (2000), pp. 383–386.

[95] Tiwari, V., Malik, S., Wolfe, A., and Lee, M. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing* (August/September 1996).

[96] Vijaykrishnan, N., Kandemir, M. T., Irwin, M. J., Kim, H. S., and Ye, W. Energy-driven integrated hardware-software optimizations using simplepower. In *ISCA* (2000), pp. 95–106.

[97] Viola, P., and Jones, M. Rapid object detection using a boosted cascade of simple features. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Dec. 2001).

[98] Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., and Agarwal, A. Baring it all to software: Raw machines. *IEEE Computer 30*, 9 (1997), 86–93.

[99] Wald, I., Benthin, C., Wagner, M., and Slusallek, P. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (September 2001), 153–164.

[100] Wang, H.-S., Peh, L.-S., and Malik, S. Power-Driven Design of Router Microarchitectures in On-Chip Networks. In *Proceedings of MICRO-36* (December 2003).

[101] Whitted, T. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (1980), 343–349.

[102] Wiskott, L., Fellous, J., Kruger, N., and Malsburg, C. Face Recognition by Elastic Bunch Graph Matching. Tech. Rep. 96-08, Ruhr-Universitat Bochum, April 1996.

[103] Woop, S., Schmittler, J., and Slusallek, P. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444.

[104] W.Zhao, Chellappa, R., and Krishnaswamy, A. Discriminant analysis of principal components for face recognition. In *International Conference on Face and Gesture Recognition* (April 1998).

[105] Yehia, S., Clark, N., Mahlke, S., and Flautner, K. Exploring the design space of lut-based transparent accelerators. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems* (2005), pp. 11–21.

[106] Young, S. Large vocabulary continuous speech recognition: A review. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding* (Dec. 1995), pp. 3–28.

[107] Zhang, Y., Parikh, D., Sankaranarayanan, K., Skadron, K., and Stan, M. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Tech. Rep. CS-2003-05, Univ. of Virginia, March 2003.